A graphical user interface for live music coding

Trabajo de Fin de Grado en Desarrollo de Videojuegos
Curso 2020–2021

Autores
Mario Tabasco Vargas
Gonzalo Cidoncha Pérez

Director
Jaime Sánchez Hernández

Facultad de Informática
Universidad Complutense de Madrid

Una interfaz gráfica para programación de música en vivo

A graphical user interface for live music coding

Trabajo de Fin de Grado en Desarrollo de Videojuegos
Departamento de Sistemas Informáticos y Computación

Autores
Mario Tabasco Vargas
Gonzalo Cidoncha Pérez

Director
Jaime Sánchez Hernández

Convocatoria: *Junio 2021*

Facultad de Informática
Universidad Complutense de Madrid

15 de junio de 2021

Abstract

A graphical user interface for live music coding

In recent years, live music coding environments have attracted many programmers with musical roots and interests. However, current live music coding environments require a certain level of technical knowledge that might result inaccessible for amateurs that lack a solid programming background. The objective of this project is to develop an application that will allow the user to generate improvised music, but through a graphical interface that abstracts all the programming. This front-end desktop application, developed in the Unity engine, will provide a user interface greatly inspired by the visual programming language Scratch, which will allow the user to configure sequences of actions by fitting together blocks that resemble puzzle pieces. The sounds will be generated by the Sonic Pi live coding environment, as it runs a program that translates command messages coming from the front-end application into actual commands that can be performed by Sonic Pi.

Keywords

desktop application, graphical user interface, live music coding, Open Sound Control, visual programming, Scratch, Sonic Pi, Unity.

Resumen

Una interfaz gráfica para programación de música en vivo

Los entornos de programación de música en vivo (*live music coding*) han despertado mucho interés en los últimos años entre los programadores con inquietudes musicales. Sin embargo, los entornos actuales requieren habilidades no triviales de programación y pueden resultar poco accesibles para los aficionados sin esa formación. El objetivo de este trabajo será desarrollar una aplicación que permita al usuario generar música improvisada, pero a través de una interfaz gráfica que abstraiga la programación como tal. Esta aplicación de escritorio *front end*, desarrollada en el motor Unity, ofrecerá al usuario una interfaz visual inspirada en gran medida en el lenguaje de programación visual Scratch, y permitirá al usuario configurar secuencias de acciones encajando bloques similares a piezas de puzle. Los sonidos se generarán en el entorno de *live coding* Sonic Pi, donde se ejecutará un programa que traduce los mensajes con comandos provenientes de la aplicación *front end* en comandos que Sonic Pi pueda llevar a cabo.

Palabras clave

aplicación de escritorio, interfaz gráfica de usuario, live music coding, Open Sound Control, programación visual, Scratch, Sonic Pi, Unity.

Contents

List of figures

Chapter 1

Introduction

1.1. Background

1.1.1. Live coding and music

Live coding is the concept of programming *on-the-fly*, improvising the code that is being written and making changes in real time. It is usually associated with the control of light systems or the creation of graphics and music. The beauty of live coding lies in the potential to create improvised artworks through code, typically in front of an audience.

Improvised musical performances are one of the most known uses for live coding. The relationship between live coding and music comes in a natural way, as music has been related to improvisation and spontaneity all throughout history (in [3], several examples of improvisation in music are mentioned, including jazz solos, improvisation in Baroque music and Native American shamanic chants).

At the time this paper is being written, a lot of live coding environments focused in the creation of musical pieces and performances (or used for these purposes) exist. Some of the most well-known are:

- **Super Collider**, a "real time audio synthesis" environment [4]. Super Collider was not initially conceived as a live coding environment, but it provides a platform for audio synthesis and algorithmic composition that can be used by performers or as a basis for other live coding languages.
- **TidalCycles**, an open source tool that allows to generate musical patterns with code [5]. It is one of the most common languages used at *algoraves* (algorithm and rave), events where people can dance to live coding performances and music generated from algorithms in general.
- **FoxDot**, a Python-based language and editor for live music coding that communicates with the SuperCollider synthesis engine to make music [6].

- **Sonic Pi**, a live coding environment developed by Samuel Aaron in collaboration with the Raspberry Pi Foundation with educational purposes [7].

- **Pure Data**, an open source visual programming language for multimedia works [8]. It is commonly used as a sound synthesis and sequencing tool for creating electronic music.

1.1.2. Visual programming languages

In recent years, a trend has appeared that consists on the development of programming languages that can be used to code by interacting with different elements in an editor's user interface instead of writing actual code. This visual programming languages are often used for tasks such as the implementation of graphic shaders in applications with real-time graphics or scripting specific behaviours. The use of a visual environment for these tasks allows less technical professionals, like designers and artists, to implement specific components of an application without the intervention of a programmer.

The two main approaches that are being used today are: node based systems (e.g., Unreal Blueprints for visual scripting [9], Blender nodes for creating materials [10], etc.) and puzzle-like applications (e.g., Scratch [11], App Inventor [12], etc.)

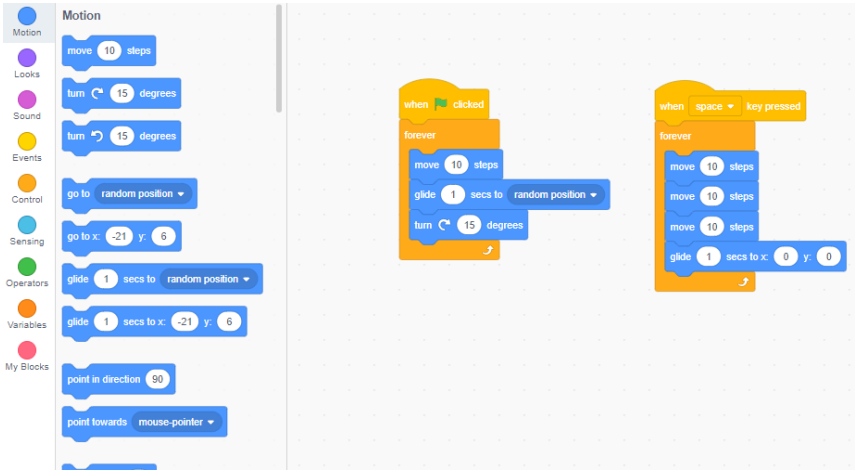

Figure 1.1: An example of code using the Scratch web application.

Scratch is probably the most known puzzle-like visual programming language. It was created as an educational tool to teach students with very low programming knowledge the basics of coding, using a colourful graphical user interface that represents sequences of simple commands as blocks that can be attached to one another. Figure 1.1 shows an example in which two *forever* loops are defined, one is executed at the start of the program and the other is activated when the user presses the *space* key. The instructions placed inside these loops are executed sequentially from top to bottom.

1.2. Motivation and objectives

The motivation for this project comes from the absence of live coding software dedicated to less technical users, in addition to the interest in developing an application with an educative purpose and a focus in drawing more people to the live coding community, regardless of their technical background.

Live coding offers lots of creative capabilities with the enormous catalog of different languages destined to different art schools, such as graphic media or music creation, but lastly, it requires some advanced coding knowledge that is not always at disposal of the public interested in artistic creation.

Nonetheless, live coding has also been used as an educational tool for introducing computational and music concepts (in [13], a discussion is made on the role that live coding could play in education). One advantage of live coding is that results can be immediately perceived when programming, which allows the programmer to approach the act of coding in a more interactive and satisfying way. Instead of introducing the basics of coding and computer science with more complex programming languages, some live coding languages can be easier to learn, or at least, more appealing to students.

Due to the difficulties given by the fact that the user has to know different coding languages in order to create media using live coding, the idea behind this project is giving the less technical users a more visual, understandable tool that lets them develop live music with an interface that doesn't require previous advanced programming knowledge. This will ease the way for these novice users to understand music creation and its technical concepts.

This application will serve as a tool to offer users with different profiles an entrance to the live coding community, from the absolute beginners to the more experienced creators. So, in conclusion, our main objectives are the following:

- Provide users with every basic functionality a live coding environment should offer.
- Bring all these features into a more clear, visible and understandable interface while maintaining the essence of live coding.
- Develop an application that allows users to create artworks that feel different and unique in every session.

Fundamentally, the main objective is to provide a different approach to live music coding that may attract less technical users and introduce them to this concept. To achieve this, we plan on creating an application that works as a visual live coding environment with a simplified functionality for users to experiment with and produce live musical performances. While the application will count with a fixed amount of features (unlike code-based live coding environments), we expect users to be able to create unique works and unleash their creativity and musical skills to deliver enjoyable performances.

1.3. Work structure

The contents of the work are structured as it follows:

- Chapter 2 serves the reader as an introduction to the application developed for the work and how to use it, without giving technical details about its implementation.
- Chapter 3 gives a detailed explanation of the software and libraries needed for the development of the application. The objective of this chapter is to introduce the reader to concepts relative to these technologies that will be mentioned when talking about how the application was implemented.
- Chapter 4 provides all the technical details about how the application and the Sonic Pi program were implemented, as well as how both programs communicate with each other.
- Chapter 5 enumerates the contributions made by the authors of this work.
- Chapters 6 and 7 provide an analysis of the results achieved with this project in contrast with the objectives that were initially established, as well as talking about the features that could be included or implemented in future versions of the application.

Chapter 2

Application User Manual

This chapter describes the application's main features and how to use them. The chapter also includes a step by step guide on the installation process and a detailed description of the user interface elements.

2.1. Installation

First of all, it is necessary that the user has the **Sonic Pi** environment installed for Windows 10 (the desktop application can be downloaded from Sonic Pi's main website: `https://sonic-pi.net/`).

The next step is to install **Sonic Pi Controller**. An installer can be found in the following website: `https://mariotab28.github.io/Sonic-Pi-Unity-Controller/`. The user can select the destination folder during the installation process (figure 2.1).

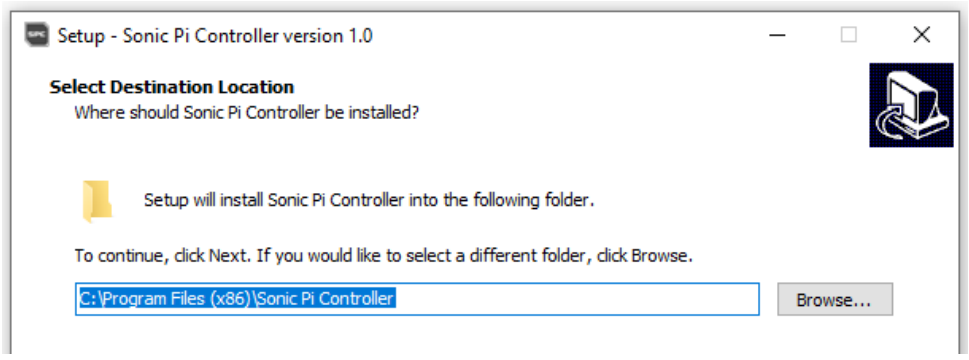

Figure 2.1: A screenshot from Sonic Pi Controller's installer.

It is recommended to copy the destination folder's path, as it will be needed in the following steps.

After going through the installation process, the user may open the Sonic Pi desktop application and load the `Launcher` file by clicking the *load* button inside Sonic Pi (figure

2.2). This file can be found in the destination folder, together with Sonic Pi Controller's executable file.

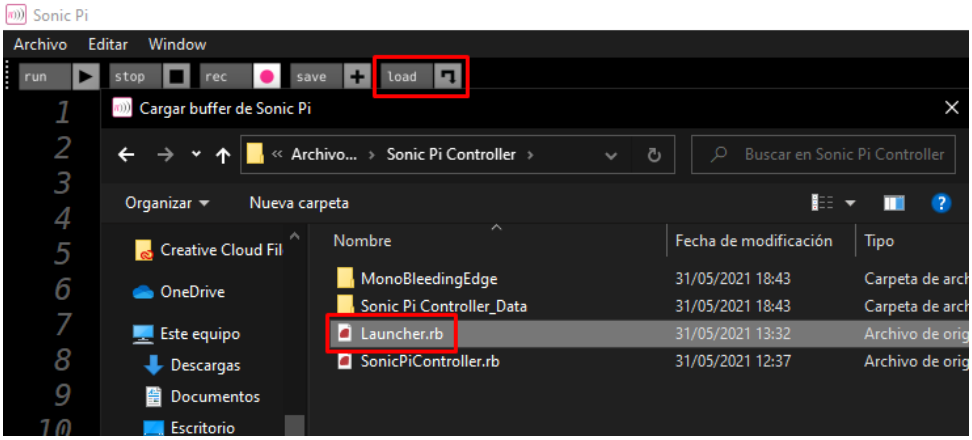

Figure 2.2: A screenshot showing how to load the `Launcher` file inside Sonic Pi.

With the launcher loaded, the next step would be to enter the destination folder's path, as it is explained in the instructions from the `Launcher` file, as it is shown in figure 2.3.

Figure 2.3: A screenshot of the `Launcher` file loaded in the Sonic Pi editor.

Then, the user can start the Sonic Pi listening program by clicking on the *run* button or by pressing the `ALT + R` shortcut. With the program running in Sonic Pi, and if the path to the destination folder is correct, the user is now able to open up the Sonic Pi Controller application (*SonicPiController.exe*) and start creating music with it.

2.2. Understanding the interface

Let's take a look at the main interface of the application.

On figure 2.4, we can see three different elements. Firstly, element A shows two buttons, the `Run Loops` button and the one with the `Pause` button. When the first one is pressed, all changes made to loops or the blocks contained in them are executed. The following changes to existing blocks, as well as the addition or removal of new blocks or loops won't take effect until the `Run Loops` button is pressed again.

The button on the right, the `Pause button`, stops the running loops at the end of their current iteration, that is to say, after clicking on the `Pause` button, all existing loops will

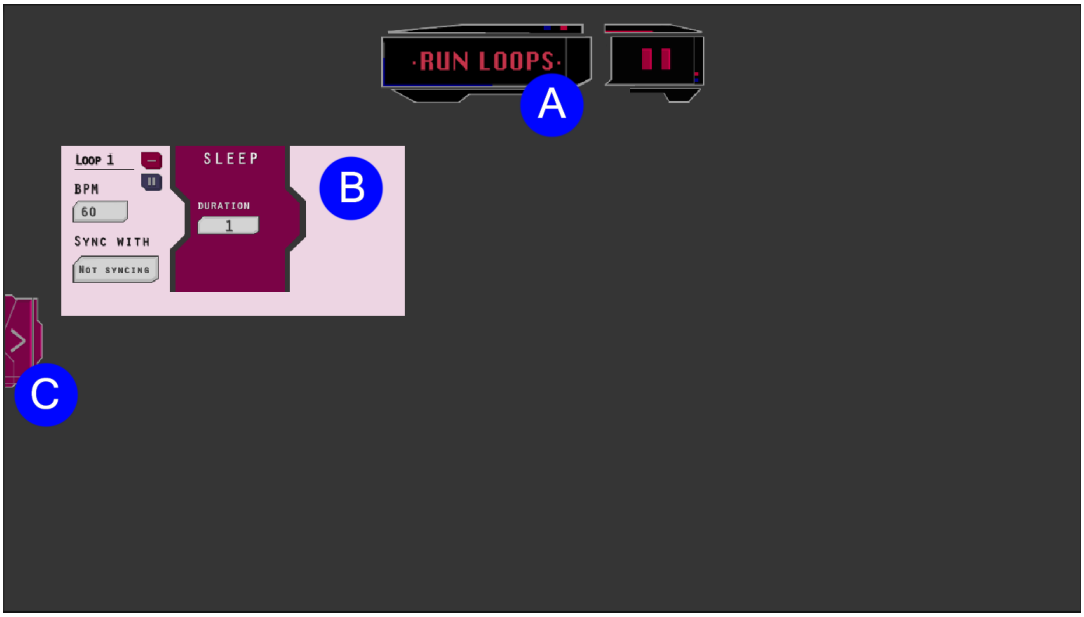

Figure 2.4: Screenshot of the first elements the user can see in the application.

reach the end of their block sequence and stop there, not starting a new iteration.

On element B, we can see a first example of the main functionality of the application, a loop. Beginning from this loop and continuing down, we will be able to place up to ten loops vertically, each one with its own elements stacking horizontally to the right. The way these elements are added to the loop will be explained in the next section of the chapter.

Lastly, element C shows an arrow pointing right. Clicking on it will bring up a menu showing the different options the user has to add to the existing loops, or add a new loop.

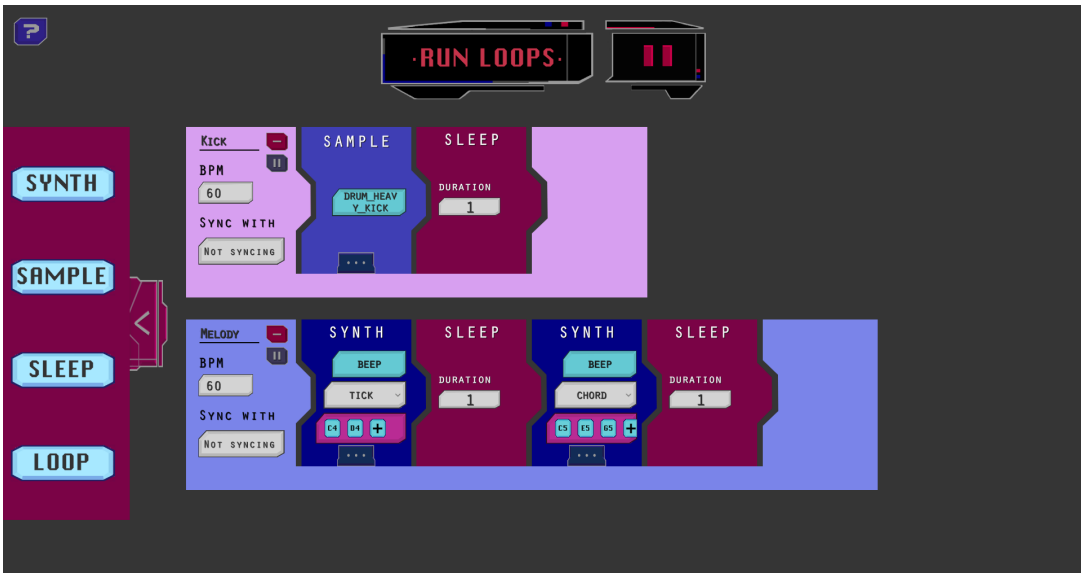

Figure 2.5: Screenshot of the application with the element menu displayed.

The element menu, shown in figure 2.5, shows four different options. The first three are elements that can be added to loops just by dragging them to the one the user wants

to add it to, and the last one lets the user add a loop to the list on the center of the screen.

2.3. Creation of a basic loop

As mentioned in the last section, clicking on the last option on the element menu and dragging the mouse to the bottom half of the screen will create a new loop.

All new loops added will appear under the last one, and the order can not change. When adding a number of loops that exceed the size of the screen, a slide bar will appear on the right side, indicating that the list can go further below, so the user can use the mouse wheel or click and drag on the screen to see the rest of the loops that have been added.

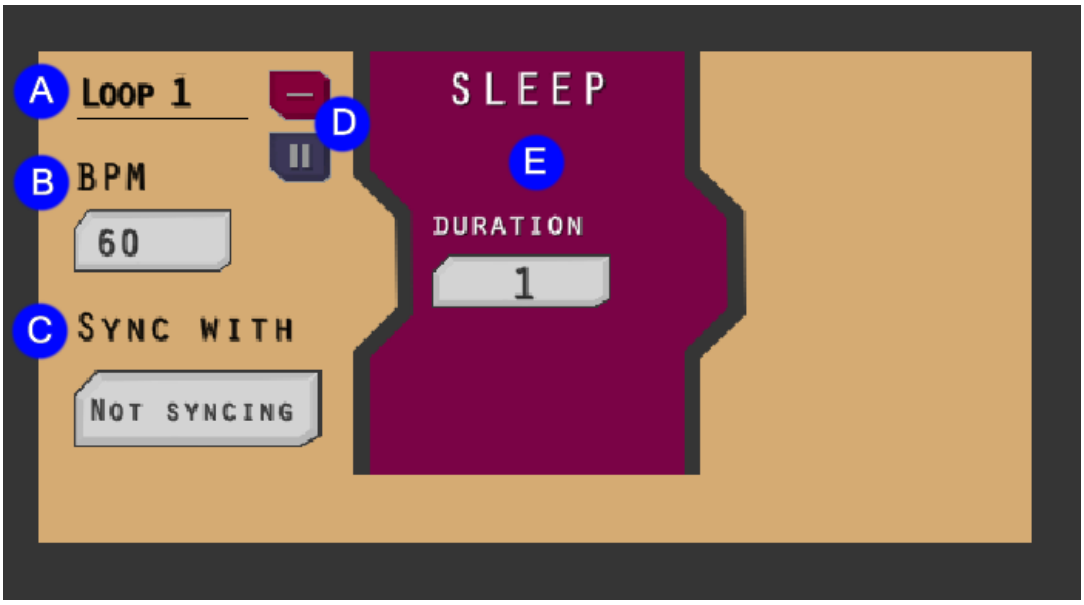

Figure 2.6: An example of a basic loop.

Looking at figure 2.6, we can differentiate the components on a loop. Firstly, element A is the name of the loop, which can be changed by the user to match the purpose of the loop or to give it a more identifiable name.

Element B corresponds to the field that sets the BPM (beats per minute) of the loop. This BPM value have an impact on the speed at which the loop will run through an iteration, as it determines the amount of time that sleep commands stop the execution of the loop. For instance, with a BPM of 60, a sleep block with duration 1 would stop the loop for one second, but the same sleep block placed inside a loop with its BPM set to 120 would cause a stop of half a second instead.

As each loop can use a different BPM value, there comes a problem in the synchronization of loops. Two loops with the same amount of elements but with different BPM each will cause the one with higher BPM to end sooner that the one with lower BPM, also making the first one start a new iteration before the second one. To avoid this, the user can set another loop to synchronize with thanks to the drop-down menu shown at element

C in figure 2.6.

In element D of figure 2.6, we can see two buttons. The one on top serves as a *delete* button, completely erasing the loop from the list. The bottom one sets the loop on pause, stopping it from looping after the user run the loops. Clicking on it again reactivates the loop.

Lastly, element E indicates the space where the blocks are going to be placed. It is important to remark that every loop must have a `Sleep Block` with a minimum duration, or be synchronized to another loop with one. This will be further explained in section 3.3.2.

2.4. Blocks

As seen in the element menu in figure 2.5, there are three types of blocks the user can add to a loop. Let's talk about each one of them, explaining each of their features.

2.4.1. Sleep Block

The `Sleep Block` is the simplest of the command blocks. It adds a stopping interval in the loop, determined by the `duration` attribute of the block, which the user can configure.

The `duration` of the block is the number of beats the loop is going to go silent. Depending on the BPM set by the loop, this number will translate into a longer or shorter silence interval.

As mentioned before, every loop must have a `Sleep Block` at the end of its elements, rule given by the basic functioning of Sonic Pi.

2.4.2. Synth Block

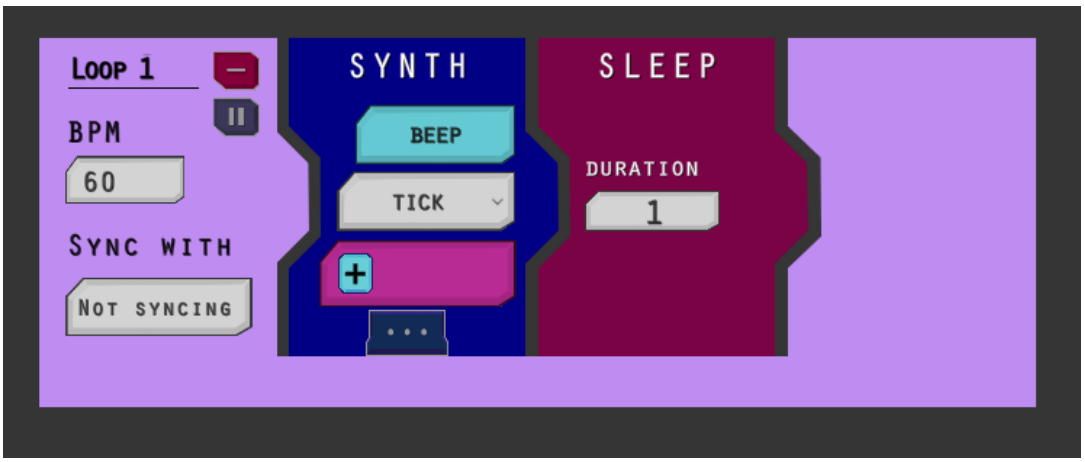

Figure 2.7: Basic `Synth Block` in a loop.

Now, the `Synth Block` has more elements to show. Going from top to bottom in the `Synth Block` seen in figure 2.7, we can firstly see the *instrument field* with the default *Beep* instrument. Clicking on it will open up a menu with all the different types of instruments Sonic Pi provides.

Just below the instrument field lays the *mode menu*. It is a drop down menu of three elements, the three modes a note or a series of notes can be played throughout and iteration.

- **Tick.** The default mode, `tick`, plays a note each iteration following the order that is established in the notes list. When it reaches its end, it goes back to the start of the list.
- **Chord.** In `chord` mode, the Synth Block simultaneously plays every note placed in the list.
- **Choose.** `Choose` mode plays just one note every iteration of the loop, choosing randomly from all of the notes in the notes list.

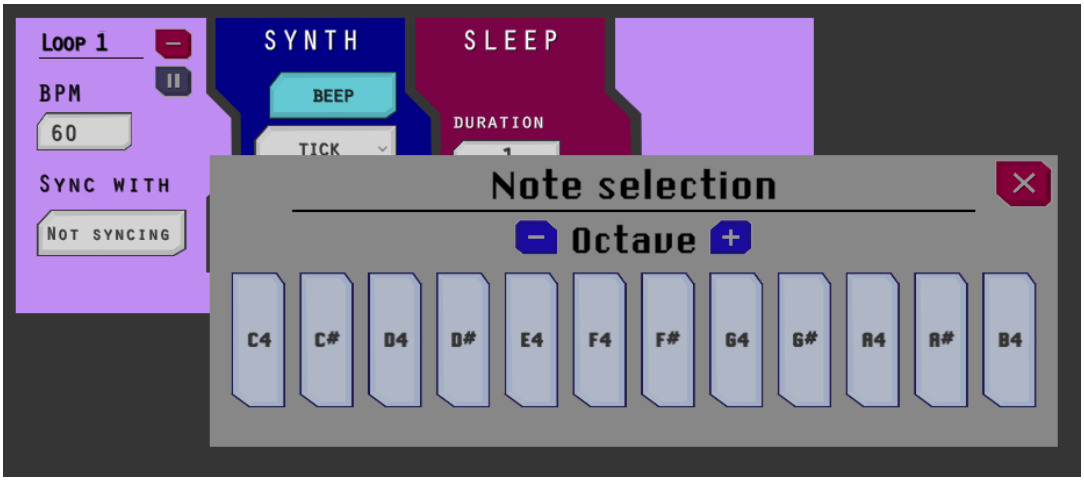

Figure 2.8: `Note Selection` panel with the default octave and the twelve notes on an octave.

The third element in the `Synth Block` is the list of notes it will reproduce. Initially, the list will be empty, but if the user click on the button with the Add symbol, an element will appear on the list with the C4 note assigned to it.

Clicking on the button will spawn the *note selection panel* shown on 2.8. With this panel, the user can change the musical note from the twelve possible notes in an octave and the number of the octave, from 1 to 8.

2.4.3. Sample Block

Lastly, the `Sample Block` reproduces prerecorded sounds instead of synthesized notes, such as percussion sounds, ambient sounds or other miscellaneous sounds. They are set sounds, so there are no further options within each sample.

Clicking the button in the middle of the block lets the user change the sample sound, choosing from a list of possible categories and the list of possible Samples in each category.

Both the `Sample Block` and the `Synth Block` have an additional button on the bottom of the body. This button lets the user change some general attributes, such as the amplification, the pan or the attack time of the sound it produces. These attributes are the same within each block, and similar between the synth and the sample blocks, this last one having some more options due to the lack of additional options in the block and the potential complexity of some samples.

2.5. Recording a session

It is possible to record the audio that emerges from Sonic Pi by pressing the Record button on the top of the Sonic Pi window. It will record everything that surges from the moment of pressing the button until the user presses it again. After stopping the recording, the user will be able to choose where to save the audio file, which will be saved as a WAV format file.

Chapter 3

Technologies

This chapter gives a detailed description of the software and libraries used in the development of the application. The application consists of a front-end desktop application that communicates with a back-end program that generates the sounds. On one hand, the main program, developed using the Unity engine, serves as the user interface to create and modify the loops of sound that will be executed by the other piece of code, which is running on Sonic Pi. The communication between both programs takes place thanks to the OSC (Open Sound Control) protocol.

The chapter is aimed at those readers who need an introduction to the OSC communication protocol, as well as the two main software products used for the development of this work: Unity and Sonic Pi.

3.1. Open Sound Control protocol

When working with software used for producing or editing sound or music, such as digital audio workstations (also known as DAWs) or live coding environments, it is often the case that a communication between two different programs or machines is needed in order to send and receive specific instructions such as what notes to play or when to start and stop recording sound. With this situation in mind, several communication protocols have been developed to be used specifically in musical and sound editing software and hardware. The most common protocols to be used in multimedia systems are MIDI and OSC.

MIDI was originally invented for communication and control between electronic or digital musical instruments. These days it is widely used among digital audio workstations to record music and sounds, allowing users to manipulate sounds in ways that would not be possible by just recording the audio of an instrument using a microphone.

On the other hand, OSC was developed to present a wireless alternative to MIDI, providing a "protocol for communication among computers, sound synthesizers and other multimedia devices that is optimized for modern networking technology" [14]. The most

common protocols used to transport OSC messages over the internet are UDP and Ethernet.

As a message-oriented protocol, OSC works with two types of packets: *messages* and *bundles* (which contain multiple messages). The data contained in these messages can either be numbers or strings. The protocol uses "native machine representations of 32-bit or 64-bit big-endian twos-complement integers and IEEE floating point numbers" [14] for numeric data. In the case of strings, they are represented as a succession of ASCII characters each followed by enough null characters to fill the 4-byte limit of data size.

Messages also include a symbolic address and a string with the message name. OSC protocol allows to address messages to specific components of objects, using a notation similar to a URL (e.g., `sonicpi/loopcontroller/1/synthplayer`).

The Open Sound Control protocol results in a very convenient way of communicating multimedia applications through data packages. Furthermore, the fact that most music-oriented software (such as SuperCollider and Sonic Pi) provide already implemented systems to handle the reception and sending of OSC messages makes this protocol a very suitable option for achieving this work's objectives.

3.2. The Unity engine

The term *game engine* is often used to name a set of tools that provide developers the environment needed to produce video games, simulations, or any kind of software that displays graphics in real-time. Some of the tools that game engines tend to include are: a graphics engine, a physics engine (to handle the movement and collisions of the objects), and libraries for the handling of the user's input events, among others.

Unity Technologies released the initial version of Unity[1] in 2005 [15], and it has been updated periodically since then, each year adding extra functionality and improving the systems already included in the engine.

The engine follows a component-based, object-oriented programming design, and was written in the C++ language, but with a .NET programming interface exposed to allow developers the use of different languages to write the scripts for their games [16]. Originally, the three supported scripting languages included C#, UnityScript [17] (a proprietary language syntactically similar to JavaScript) and Boo. However, only C# is officially supported today, as the compilation of Boo and UnityScript became deprecated in 2017 and 2018, respectively.

One of Unity's main features is the **Unity Editor** (figure 3.1), which serves as the engine's development environment and allows developers to configure their game scenes and objects, visualize the modifications that they make, interact with some of the engine systems and deploy a build of the game for one of the multiple platforms that Unity supports.

[1] Unity official website: `https://unity.com/`

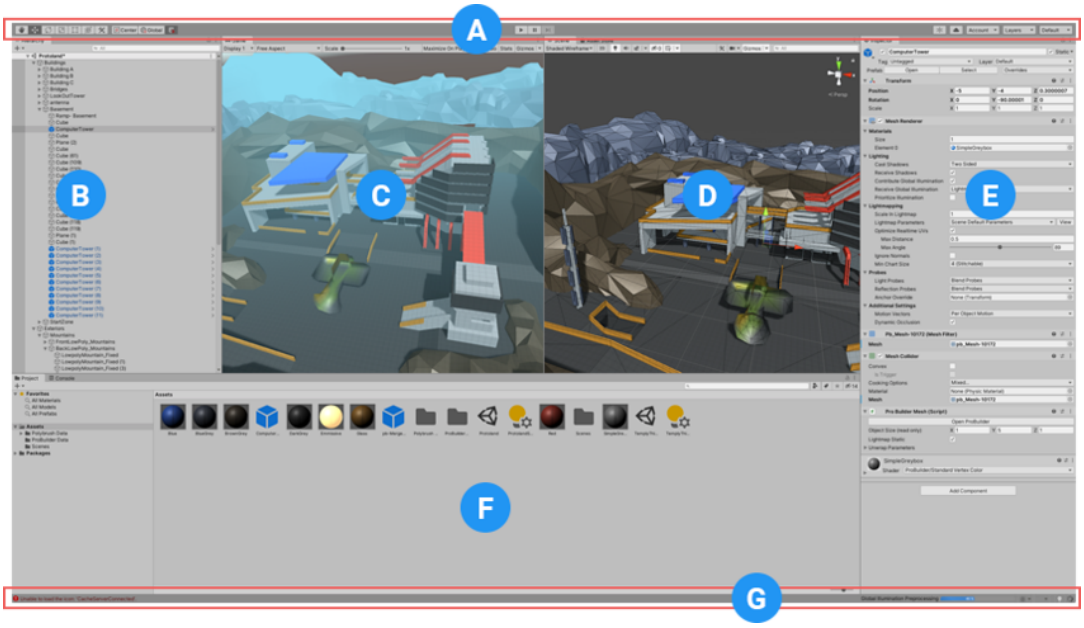

Figure 3.1: Screenshot showing the Unity Editor and some of its sub-windows, taken from the official Unity documentation [1].

3.2.1. Unity User Interface framework

One of the systems contained in the Unity engine is *Unity UI*, the Unity user interface system, or as it is described in the official manual: "a UI toolkit for developing user interfaces for games and applications" [18]. This interface system has been of special interest when considering what was the best option for the development of this work, as the goal was to implement a visual interface to interact with the live coding program, which means that a substantial portion of the development time was going to be spent in the implementation of the user interface and its individual elements.

This user interface toolkit offers a set of fully-configurable and already implemented elements for the interface, which can fulfill the needs of most of the games and applications that are developed using Unity in regards to the design and implementation of their user interface. Some examples of these elements that were used in the development of this work's main application are: buttons, input fields, scrollbars, drop-down lists, layout groups, etc.

All these objects are contained in the scene's *canvas*, the area dedicated to the display of the interface objects. The order of the elements inside the canvas hierarchy determines the order in which they will be rendered, which affects whether an interface element will be displayed on top or behind another. The scene's object hierarchy may be consulted by looking at the hierarchy window in the Unity Editor (letter B in figure 3.1).

The properties of the element in relation to its position on the screen, rotation and scale, as well as its anchors and pivot, are determined by the *Rect Transform* component, that all canvas elements use instead of the regular *Transform* component, which is present in objects outside of the canvas.

Figure 3.2: Screenshot of the Unity Editor showing the Rect Transform component of an interface object, as it is displayed in the inspector window, taken from the official Unity documentation [2].

TextMesh Pro

TextMesh Pro [19] is a package developed by independent developer Stephan Bouchard to replace Unity's original text-related components. It started as a purchasable asset in Unity's Asset Store, but was acquired by Unity in 2017 and has been included in the Unity Editor as an alternative to Unity's UI Text. The package not only delivers visual quality improvements in the rendering of text objects, but also gives more control to the user on the formatting and layout of the text.

UI Builder

Unity Editor version 2019.3 added a new user interface system called *UI Builder* [20], that allows developers to use UXML (inspired by the markup language XML) [21] and USS (similar to the CSS style sheet language) [22] files to visually design and edit the elements of the interface inside the Unity Editor. The main goal of UI Builder is to allow less technical users, such as UI artists or designers, to build a user interface using the Unity Editor.

UI Builder can be used by adding the package to the Unity project using Unity's Package Manager.

3.2.2. UnityOSC: A package for handling OSC messages in Unity

Despite the fact that Unity does not provide a toolkit of its own for handling OSC messages, the engine is attractive enough to inspire the creation of projects that communicate with sound-related and musical software, such as controlling the game's sounds with an external program by sending OSC messages from Unity or, going in the opposite direction, using an application made in Unity to receive messages and render images according to them. This can explain the existence of numerous external libraries that can be imported into Unity to handle the sending and reception of OSC messages.

Perhaps the most known of these software libraries is UnityOSC by Jorge Garcia Martin [2]. The plugin allows the configuration of both an OSC client and server to either send or receive OSC packets and bundles, abstracting the most network-related part of the code so that the programmer can focus on the handling of OSC data.

In this regard, UnityOSC includes classes such as OSCHandler, which works as a Singleton class that handles all client and server connections and the communication of OSC data between all of them. Users of this plugin are encouraged to modify and initialize this handler to adapt it to the specific needs of their applications.

In order to initialize OSC transmitters and listeners, it is only required to create instances of OSCClient and OSCServer classes and configure their identifier, IP address and port. After that, a list of elements can be sent to a client by calling the function `SendMessageToClient` with the client identifier and address as well as the list of elements as parameters. This creates an OSCMessage instance with the specified parameters that will be sent to the client through the UDP Internet protocol. The data types supported by OSCMessage are: int32, int64 (longs), floats, doubles, strings and byte arrays. Attempting to send data of a different type throws an "Unsupported data type" exception.

UnityOSC includes a simpler alternative to the creation of OSCServer, the OSCReceiver class. When an OSCReceiver is configured to listen on an specific port, it allows the programmer to handle the reception of incoming OSC messages thanks to a thread-safe OSCMessage queue. The application may ask the receiver if any waiting messages exist and, if that is the case, take them out of the queue one at a time to handle the data.

While UnityOSC continues to be considered as a viable way of handling OSC messages, it has been recently announced by the plugin's creator that its development and maintenance has stopped [23], and multiple alternatives (some of them based on UnityOSC code) can be used, including extOSC, OscCore or OscJack.

3.2.3. Working with JSON files in Unity

When developing games and applications that require a certain amount of configuration, whether it is to configure levels inside a game, object attributes or elements of the interface, it is a good practice to implement systems that read the configuration of these features from external files and arrange them appropriately. This allows programmers to abstract certain elements of design and configuration from the application source code, which can be useful for less technical members of the team to configure these elements without worrying too much about the internal implementation. It also improves the workflow, as this implies that recompiling the code is not needed after making changes to these configurable elements.

Multiple options exist for the format of this kind of files. The most common of these formats include markup or human-readable text formats aimed at the representation of data structures, (e.g., XML). One of the most commonly used data interchange formats is JSON (JavaScript Object Notation), which uses lists of attribute-value pairs to represent data in a human-readable text format.

These days, a lot of languages and environments have libraries that implement the

[2] `https://github.com/jorgegarcia/UnityOSC`

parsing of JSON files and the serialization of data into JSON format. Unity, for example, implements its own framework for working with JSON files, JsonUtility [24]. This module includes the methods `ToJson` and `FromJson`. As their names implies, these static functions are used to serialize and deserialize, respectively, C# serializable objects into JSON data, and vice versa.

JsonUtility includes the basic functionality necessary to work with JSON files, but its usefulness can become scarce when the user needs to work with more complex types of data, like dictionaries or nested lists. In this case, some alternative libraries may offer a more complete functionality. One of them is **Json.NET**, a "popular high-performance JSON framework for .NET" [25], developed by Newtonsoft.

3.3. Sonic Pi

Sonic Pi is an open-source code-based music creation tool and live coding environment [7]. It was developed by Samuel Aaron in collaboration with the Raspberry Pi Foundation, with the intention of introducing basic computing concepts to students with no programming background, as they experiment with the creation of musical pieces. Samuel Aaron explains[3] that, in order to teach the fundamentals of programming, using a language and environment friendly enough with beginners and, most importantly, introducing them in interesting and enjoyable tasks, proves to be a good approach for technology teachers to take at an elementary level [26].

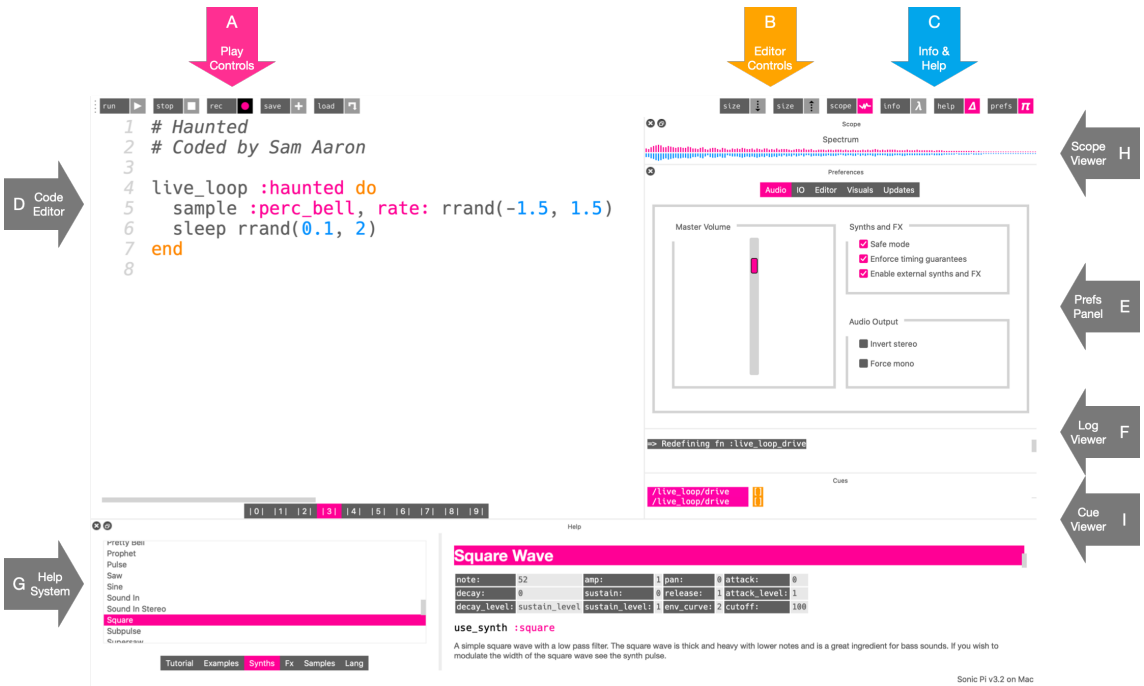

Figure 3.3: A screenshot showing Sonic Pi's text editor interface.

Sonic Pi is designed to follow this educational approach, and it achieves it thanks to a

[3]Samuel Aaron gives a detailed exposition of Sonic Pi's educational capabilities in the following interview: `https://www.youtube.com/watch?v=7sEMKXrRaAs`

simple but effective text editor and a straightforward programming syntax.

The following piece of code may help to illustrate Sonic Pi's syntactic simplicity:

```
live_loop :synthesizer do
    play 60
    sleep 1
end

live_loop :bass do
    sample :bd_haus
    sleep 0.5
end
```

In the example above, we can see two different loops that will be executed concurrently, the first of them reproducing note 60 (C4 note in Sonic Pi notation) and sleeping for one beat[4]. The second plays one of the multiple Creative Commons licensed samples that are included with Sonic Pi, and then sleeps for half a beat. This simple piece of code could be used to introduce students to the concept of concurrency in computing [27], and it provides an insight into the endless possibilities that Sonic Pi may offer, both to teachers and musicians.

The programming language used to code in Sonic Pi is a variant of Ruby [28], developed specifically to serve as Sonic Pi's live coding language [13]. Ruby is a programming language that can be easily manipulated to build domain-specific languages (or DSL). One of the reasons given by Samuel Aaron for using a Ruby-based language instead of creating an original syntax is that it was the his desire to "introduce languages in the classroom that reflected actual commercial practice" [26], as many students seem to be interested in "learning the tools that real programmers use".

Internally, Sonic Pi makes use of the *SuperCollider* synthesis engine [4], which keeps running in the background as a server process while Sonic Pi is being executed, generating the sounds required by Sonic Pi after interpreting the user's code.

3.3.1. Deterministic execution in Sonic Pi

Another important fact about Sonic Pi to take into account is its goal to ensure a deterministic execution of the code, meaning that the same piece of code must produce the same sounds, regardless of the time of execution or what machine is being used. This design decision comes with some consequences. For example, the user of Sonic Pi may want to use a random function to generate a different value in each tick for a note or attribute, adding a certain amount of variety or unpredictability to its creation. However, users must bear in mind that the sounds produced by Sonic Pi will be the same each time they press the *Run* button, and other people will get the same results if they execute the same code in their respective machines.

One more consequence of Sonic Pi's deterministic approach is that some Ruby func-

[4]The amount of time the loop is sleeping with the *sleep* command is determined by the BPM (Beats Per Minute), with 1 beat being equivalent to a second with a BPM of 60.

tionalities which may not guarantee this level of similarity in the execution of the code are not officially supported by Sonic Pi, therefore their correct functioning is not assured[5].

3.3.2. Time handling in Sonic Pi

When creating musical pieces with Sonic Pi, the handling of time may be of concern for users with a minimum of musical knowledge. As is well known, timing and duration of both sounds and silences are key concepts in any piece of music. However, when working with computers the notion of time can get difficult to handle, as we must take into account not only the time at which a task is supposed to be executed, but also the time it takes for the machine to execute that task. This can get specially complicated when several instructions must be performed at the same time (which is physically impossible for a computer to achieve).

In [29], we can find a technical description of how the time behaviour of Sonic Pi works.

```
play :C; play :E; play :G
sleep 1
play :F; play :A; play :C
sleep 0.5
play :G; play :B; play :D
```

For the code example above, the expected execution would be the following:

1. First, notes C, E and G are played together simultaneously. Then, one second of silence is expected.
2. After completing the sleeping time, the next set of notes (notes F, A and C) are played at the same time, and then another sleep command is executed, this time with a duration of half a second.
3. Finally, after waiting for half a second the last three notes (G, B and D) are played together.

While this would be the expected result, what actually happens is that each instruction takes some time to execute, which varies from one computer to another, and that has an impact in the timing of the events being executed. The first thing that this implies is that the three sets of notes from the example will not play all notes at once, as there will be a small delay between the *play* commands, corresponding to the execution time of that instruction.

In order to correctly handle the elapsed time accumulated between sleep calls, Sonic Pi uses a redefined version of Ruby's sleep command. The original sleep command works as it would in any programming language of general purpose, expecting an explicit time value,

[5] In this post on the official Sonic Pi forum, a member of the Sonic Pi Core Team mentions that the use of custom classes is not possible using Sonic Pi's officially supported syntax: `https://in-thread.sonic-pi.net/t/is-it-possible-to-make-custom-classes/3632/25`.

which indicates the exact time that the program's execution will be stopped. In Sonic Pi's implementation of sleep, the command also takes a duration value t, but actually sleeps for the implicit expected value, that is to say, the time needed for at least t seconds to pass since the last sleep command.

To achieve this effect, Sonic Pi adds a thread-local variable, which represents the *virtual time*, that is only advanced with a sleep command and is used to schedule the execution of sound instructions. Therefor, in order to make the execution of the program follow the timing requirements, the sleep command considers the elapsed time since the last sleep (Δ_T), and appropriately reduces the requested duration (Sr) of the sleep real time (St).

$$St = Sr - \Delta_T$$

While the scale of the three time spans after each note playing from the previous example is not big enough for the listener to notice, one may wonder what would happen if, instead of having a set of three notes, the computer was asked to perform a sequence of commands big enough to overrun the sleep time. In this respect, another thread-local variable, *scheduleAheadTime*, is used, which adds a constant value to the virtual time of the executing events. When the time taken to perform all commands between two calls to sleep is longer than the sleep time, the thread is considered to be falling behind, and an explicit timing warning is displayed to warn the user. Then, if the thread falls further behind (the user can change the amount of time before this condition is met), the thread is stopped by Sonic Pi throwing a time exception. This is specially useful to stop the overrunning thread from permanently consume resources when this kind of exception is caused in a loop. One direct consequence that this has in the use of Sonic Pi is that, when creating a loop of sounds, the user must make sure that the time slept between loop iterations is never lower than the time taken to perform those iterations.

3.3.3. Live loops

When performing a live coding session, one of the basic needs of the user will be to introduce a sequence of sound and sleep commands playing in an endless loop. The *loop* command allows to do just so, by repeating the sequence of commands contained in the loop until the user tells Sonic Pi to stop the execution.

However, this loop command on itself does not fulfill another basic need of live coding performers: the concurrent reproduction of sounds contained in different loops. The user might want to play, for example, a bass line and a sequence of drum sounds accompanying the main melody, but the manual intertwining of the sounds can get incrementally complex as the number of elements increase. This is why playing multiple loops at the same time could come in handy. When programming two consecutive loops in Sonic Pi, the user is telling the application to run an endless loop and then run a second one, but the latter is never reached as the main thread is stuck in the first loop.

One solution can be to wrap at least one of the loops inside an *in_thread* command. This command create a new Sonic Pi thread (which is a written on top of a Ruby thread, but works in a similar fashion), so that the instructions inside the in_thread body will be

executed in this thread and not in the main one [30].

```
in_thread() do
  loop do
    use_synth :prophet
    play chord(:e2, :m7).choose, release: 0.6
    sleep 0.5
  end
end

in_thread() do
  loop do
    sample :elec_snare
    sleep 1
  end
end
```

After wrapping all the loops inside in_thread commands, the user will finally listen to all these loops playing simultaneously, but this solution comes with an issue that comes up when the user decides to add something else and hit the Run button without stopping the previously running execution (which can be expected in a live coding session). When doing so, the user will hear how the threads that were already running are duplicated. While the expectations might be that only the additions are executed when the Run button is pressed, the same threads are being created in each run. This can be prevented by giving each thread a name, using the following syntax:

```
in_thread(name: :thread_name) do
    ...
end
```

When running the program, Sonic Pi will not create new threads with the same names as already running threads, which prevents the issue with duplicating threads.

At this point, the user is already able to add several loops that will play concurrently, run them to start playing their sounds, add a new loop and run the code again without creating duplicated threads. But there is one more demand from live coding users that the application must fulfill: to allow the modification of loops already in execution without stopping the run.

With the previous solution, Sonic Pi would ignore the modification of a loop unless the name of its thread is changed. But changing the name is not an option either, as this would be the equivalent of creating a new thread without telling the previous loop to stop. The answer is to move the piece of code that will be played from the loop into a function, and then call that function inside the loop. When running the code after modifying the function, the thread that contains the loop will not be duplicated, as it has the same name, but it will behave different as the function that is being called inside the loop has changed.

```
define :my_sound do
  play 60
  sleep 1
end

in_thread(name: :looper) do
  loop do
    my_sound
  end
end
```

Although this last solution works perfectly fine, it could be consider as rather complex if we look at it from the perspective of novice programmers, as they are expected to be familiar with concepts such as threads, functions and scope. To simplify the way a user can create a loop that suits the needs of a live coder, Sonic Pi API includes a special kind of loops called ***live loops*** [31]. Live loops help abstract the previously discussed implementation of live coding loops, allowing the user to create a modifiable loop that runs on its own thread by writing a single instruction to wrap the body of the loop.

```
live_loop :foo do
    play 60
    sleep 1
end
```

This syntax proves to be simpler and more appropriate for Sonic Pi educational motivation.

As it happens with threads, live loops have names, and it is not possible to create two of them that are called the same. The name can be used to identify different loops, for example, when synchronizing one loop to another.

When two loops start running at different times (i.e., one loop is added after another started running, then the user runs the code one more time), there is a chance that these loops will sound out of sync. While this behaviour is not necessarily a bad thing, it is important to let users decide whether they want specific loops to be synchronized. To achieve this, live loops in Sonic Pi can synchronize to a different loop using the **sync command**.

For example, let's assume that live loop A is synchronized with live loop B. Live loop A will perform its commands until reaching the `sync "/live_loop/B"` command. Then, it will wait until live loop B reaches the end of the loop. The way one loop tells the others its iteration has finished is by sending a `cue` message containing its name, and all live loops do this by default. If the user decides so (by setting `bpm_sync` option to true), live loop A will also inherit the BPM of live loop B. After receiving the corresponding `cue` message, live loop A continues its execution until reaching another `sync` command. If live loop A is sleeping at the same time that live loop B sends the `cue` message, it will continue its execution normally until reaching the `sync` command.

3.3.4. OSC messages in Sonic Pi

Sonic Pi is already configured to listen to port 4560 by default for OSC messages coming from the same machine it is executed on.

The following piece of code, taken from Sonic Pi official tutorial [32], illustrates how to receive OSC messages in Sonic Pi.

```
live_loop :foo do
  use_real_time
  a, b, c = sync "/osc*/trigger/prophet"
  synth :prophet, note: a, cutoff: b, sustain: c
end
```

The example shows how to listen to an OSC message and use its data to configure some attributes (note, cutoff and sustain) for a synth player. The syntax may resemble the synchronization within live loops, using the `sync` command to wait for a message to appear, blocking the execution in the meantime.

To send OSC messages out of Sonic Pi, the destination IP address and port can be configured using `use_osc "localhost", 7000`, where *localhost* would be the host and *7000* the port. Then, it is as simple as using the `osc` command and specify the content of the message: `osc "hello/world"`. The previous code would the following message: `/osc:127.0.0.1:4560/hello/world`. The "/osc" prefix is added to the message automatically by Sonic Pi.

Chapter 4

Implementation

This chapter will serve to describe the implementation of the application and all its features, explain some of the decisions made on the programs architecture and report some issues found during the development process and how they were addressed.

The chapter has been split up into two main blocks: implementing the program that handles Sonic Pi execution of the sounds and the development of the main application using the Unity engine. A third section at the end of the chapter describes the process of deploying a build for the application.

4.1. Architecture of the program in Sonic Pi

As the main objective of this work was to develop a graphical interface for live coding performances, it was decided that the handling of sound, time and loops should ultimately lie with a live coding environment that already implement these functionalities. For this purpose, we decided to use Sonic Pi. As it was explained in chapter 3, Sonic Pi makes use of its own domain-based variation of the Ruby language, specifically developed for the purpose of delivering a simplified live coding syntax that could be used to teach music and computing concepts at an elementary level.

The Sonic Pi program is implemented in a single source file (*SonicPiUnityController.rb*), and its code can be divided into four blocks: definitions of message and attribute structures, the listening thread, the processing thread and initialization of the program and threads. These blocks will be further explained in following sections.

4.1.1. Command and Attributes structures

When approaching the task of controlling a Sonic Pi execution through another program, different options where taken into consideration. However, the fact that Sonic Pi already included an implementation for the handling of OSC messages suggested that this was the way to go.

The approach taken for the implementation of this program was the following: at the beginning, a rack of **loops** (each with its own listening and processing thread) will be created; internally, each loop will consist of a sequence of commands that is going to be translated into actual Sonic Pi instructions. These commands will consist of a series of attributes needed to configure their sound and effect on the execution.

This method suggests the implementation of a hierarchical architecture of classes to define structures for Command and Attributes types. However, while the Ruby programming language supports an object-oriented programming approach, Sonic Pi does not officially embrace the creation of classes. As explained by Samuel Aaron and the Sonic Pi Core Team [33], while the user might be able to get Ruby classes to work in a Sonic Pi program, its use is not officially supported and therefor not recommended. The creation, destruction and lifetimes of objects using Ruby classes are a complex thing to handle, and they are not the same for each execution of the program on different devices. This makes it impossible to work with Sonic Pi's deterministic approach (as explained in chapter 3).

For the development of this program, Ruby structures where used, as they prove to work better with a Sonic Pi program and not cause any error at runtime. As seen in Ruby documentation, a Struct is "a convenient way to bundle a number of attributes together, using accessor methods, without having to write an explicit class" [34].

The structures that were needed for the implementation of this program are:

- **Command structure.** It represents an action command inside a loop. The command structure contains the following attributes: the index of its loop, its own index inside the loop command list and its attribute structure.

- **Loop attributes structure.** It wraps up the following attributes: a boolean indicating whether the loop is active, a string with the name of the loop to which this loop is syncing (if any) and the value of its internal BPM.

- **Synth attributes structure.** It contains the common attributes needed for the configuration of any synth play command.

- **Sample attributes structure.** It contains all the attributes needed for the configuration of any sample.

- **Sleep attributes structure.** It contains the duration that will be passed to the sleep command.

Both the `SynthAttributes` and the `SampleAttributes` structures include attributes for the amplitude (`amp`) and stereo position (`pan`) of the sound, as well as the name of the synth or sample to be played, respectively.

Additionally, synth commands count with two specific attributes needed to specify which notes to play and the playing mode.

4.1.2. Functioning of the listening thread

The listening thread is the one in charge of waiting for incoming OSC messages, parsing them into Command structures and adding these commands to their corresponding loop to apply the changes.

In order to listen for incoming OSC messages from the main application, we use the `sync` command (as explained in chapter 3) to listen in address `"/osc*/sonicpi/unity"`.

When a message is captured, it is stored in a local variable as a list of type-agnostic values. The first thing this thread does with this list is looking at the initial element, which may be a string with the code `"stop"`. If that is the case, it means that the main application has been closed, so the Sonic Pi programs must be stopped too. In this regard, a new command with the name "stop" is added to stop the processing thread. Afterwards, the listening thread is killed by calling the Sonic Pi `stop` command.

If the initial message value is not "stop", the message parsing process begins.

Content of each message

To handle the OSC messages received by the program and correctly apply the modifications that they imply on the loops, an specific format has to be used.

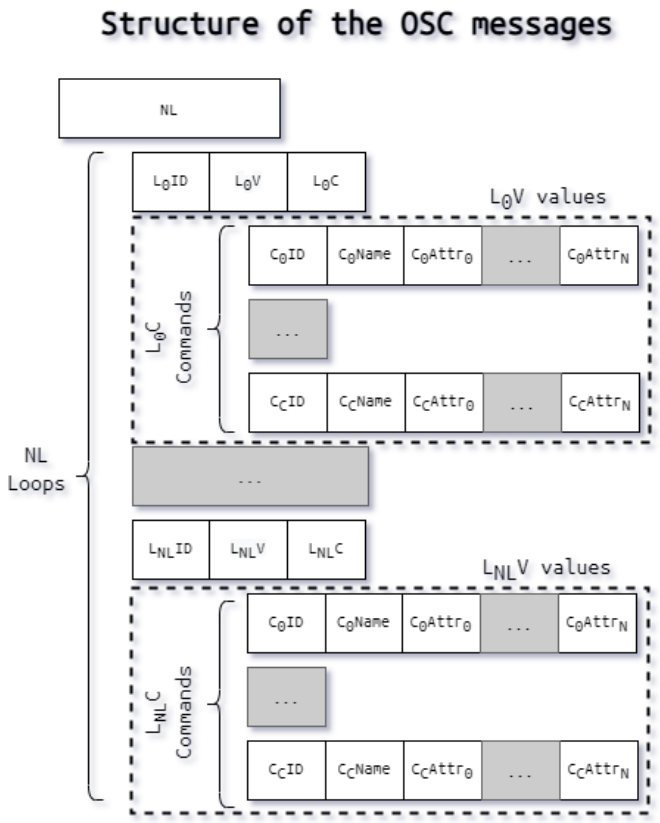

Figure 4.1: An outline of the structure of the OSC messages that can be handled by the program.

Figure 4.1 shows the structure that a message must have for it to be correctly parsed by the listening thread. The first element must be an integer number indicating the number of loops that are communicating changes from the application.

For each loop, the first three values to be read are: the loop's index (L_nID), the number of values to read for this loop (L_nV) and the number of commands (L_nC) whose attributes will be parsed for this loop. If the loop index doesn't correspond to the index of the listening thread, the number of values is used to skip all the elements of that loop so that the thread's index can be compared with the next loop index in the message.

If the index of the loop does coincide, L_nC commands are parsed. The first value read for each command corresponds to its index inside the loop's command list, which is used to either add or overwrite a command at that position in the command list. The next element, C_nName, determines the type of command (sleep, synth or sample), which indicates how to parse the next elements (a sequence of values for the specific attributes of the action), to form the corresponding command.

4.1.3. Functioning of the processing thread

For each of the loops, there is a thread running the command-processing function parallel to the listening thread. This thread is in charge of looking at the command list of its specific loop and execute a Sonic Pi command using the attributes specified by the command structure that must be processed at that moment. This thread is implemented inside a Sonic Pi **live loop** (take a look at chapter 3 for more details on live loops).

Before processing any command, the settings of the loop are checked. First of all, the loop might be configured to be inactive. If this is the case, no commands are processed at all and the thread sleeps for 1 second, as it is mandatory for loops to include at least one sleep instruction.

The synchronization between processing threads is possible thanks to them being actual live loops. If the loop is active, the processing thread consults whether it is in synchronization with other loop and waits for that loop's cue message if that is the case. Otherwise, the loop is configured to use its corresponding BPM value.

After taking a look at the loop configuration, a Ruby `each` block is opened to treat each of the commands of the loop. For every command, the type of action it represents is determined using a switch statement. In the case of synth and sample commands, a `commandPlayed` flag is set to true to indicate that an OSC message must be sent to the application indicating the reproduction of an action for that loop. To make this notification with sleep blocks, the message must be sent before performing the actual sleep, and that is why this flag is not used in this case.

As it was mentioned previously, Sonic Pi forces loops to sleep for a minimum amount of time on each iteration to avoid issues with the handling of time and sounds. To abstract the final user from this particular behaviour, the program counts with a safe mechanism that makes the loop sleep for half a beat when no sleep is being performed by the loop. One boolean flag is used to indicate if the loop has already slept and another one is used to indicate if the loop is in synchronization with another one. In the case of a synchronized

loop, it would be allowed not to sleep, as the *sleeping responsibilities* would be passed to the loop it is synchronizing with.

4.1.4. Initialization of the Sonic Pi program

When running the program in Sonic Pi, a list of operations must be performed in order for the application-Sonic Pi ecosystem to work as expected. The first thing the program does is waiting for an "initialization" message from the main application. This OSC message will contain two values: a string with the word "init" and an integer number representing the number of loops to be created at the beginning.

```
while nLoops <= 0 do
  val = sync "/osc*/sonicpi/unity/trigger"
  if val[0] == "init"
    nLoops = val[1]
  end
end
```

This number of loops will remain constant throughout the execution, instead of increasing/decreasing the loop pool when adding or removing a loop in the main application. This approach was taken to simplify the handling of the arrays of loops and commands, which started getting quite complex when implementing the removal of loops, specially when erasing loops between two other loops.

For the initialization of each loop, the loop structure (containing its settings) and its command list are initialized.

```
# Creates the rack of loops
for i in 0..(nLoops - 1)
  puts "Creating loop " + i.to_s
  commands[i] = []
  loopAttr = LoopAttributes.new
  loops[i] = loopAttr
  # Starts loop listening thread
  doListenerLoop(i, commands, loops)
  # Starts loop playing thread
  doPlayerLoop(i, commands, loops)
end
```

Afterwards, both the listening and the player threads have to be created. This is done in the `doListenerLoop` and `doPlayerLoop` functions.

```
def doListenerLoop(id,commands, loops)
  puts "Listener " + id.to_s
  # LISTENER LOOP
  in_thread do
    loop do
      # Listen commands for command list 'id'
      listenUnityCommand(id, commands[id], loops)
      sleep 0.1
    end
  end
end

def doPlayerLoop(id,commands, loops)
  puts "Player " + id.to_s
  # Set the live_loop name
  loopName = "playerLoop#{id.to_s}"
  # PLAYER LOOP
  live_loop loopName do
    # Process command list number 'id'
    processCommands(id, commands[id], loops)
  end
end
```

4.2. Development of the main application

The main application provides the graphical user interface to control the Sonic Pi program. Most of the application development workload lay in the implementation of the multiple elements that make up for the user interface, which include panels, buttons, input fields and, most importantly, loop objects and their connectable blocks.

For the development of the application, version 2020.1.13.1f of the Unity engine was used. While Unity was initially conceived as a video game engine, its use is widely spread in the development of products of a different nature. The work's application is not designed to be a game, but rather a visual tool to help users of multiple backgrounds enjoy the creative capabilities of live coding.

The application is composed of only one Unity scene, which contains the canvas and all its interface-related objects (the ones that are initially present as well as the blocks that will be added by the user) in addition to the empty objects that hold the management scripts.

4.2.1. Architecture and main components

The architecture of the main application can be divided into four blocks: the management of the OSC communication with the Sonic Pi program, the management of the list of loops and their internal array of blocks, the configuration of the visual appearance of the blocks and the code that implements the functionality for other elements of the interface.

Management of the communication with Sonic Pi

The communication with Sonic Pi through OSC messages is handled thanks to the UnityOSC plugin (see chapter 3). To allow the sending of messages from different components and objects of the application, and also to abstract the internal functioning of the UnityOSC methods, a singleton[1] class with the name of `SonicPiManager` is used. This globally accessible manager (contained in a Unity empty object present on the scene) is in charge of the following tasks:

- Initializing the OSC components needed for the handling of OSC messages (`OSCHandler` and `OSCReceiver`).
- Deserializing the configuration files which contain the information needed for an understandable communication with Sonic Pi (attributes of each block and sample/synth names).
- Sending the initialization message to Sonic Pi at the start of the application.
- Handling incoming OSC messages from Sonic Pi.
- Providing public functions for sending messages to Sonic Pi.

The definition of the classes for the representation of the message structure are also included in the `SonicPiManager.cs` file. These classes include: `ActionMessage` (the parent class for message classes), `EditLoopMessage, SleepMessage, PlayerMessage` (from which sample and synth message classes inherit), `SynthMessage` and `SampleMessage`. All these classes implement a `ToObjectList()` method which returns a list of values for a single command, following the message structure detailed in figure 4.1. The list is returned as a variable of type `List<object>`.

Management of the loops

For managing the array of active loops in the application, another singleton class is implemented: the `LoopManager`. The main tasks for which this manager class is responsible are:

- Instantiating new loops and removing the ones being deleted, as well as managing the effect of these actions in the application.
- Compiling the command messages of all the loops into the final list of values that will be sent as an OSC message, and telling the Sonic Pi Manager to do so.
- Providing public methods for the modification of existing loops from external scripts; for example, when adding, moving or removing blocks inside one loop.

The management of the blocks contained in each loop is internally handled by them through their specific component script: `LoopBlock.cs`.

[1] The singleton pattern provides a way to implement classes with unique, globally accessible instances. The use of singleton classes is wide spread among Unity users and a way to implement one in the engine is well documented [35].

Main block components

All blocks (sleep, synth, sample and even loop blocks) contain two main components:

- The **Block Shape** component (`BlockShape.cs`) is where the visual configuration of the blocks is implemented (more on this in section 4.2.5). It includes public methods for the configuration of the block (changing the color, activating or disabling the block's edge or gap, etc.).
- The **Block Attributes** component (`BlockAttributes.cs`) contain the attributes relevant for the management of the block and the generation of its command message. These attributes include: the index of the block inside its loop, a string with the name of its action and an ActionMessage instance with its message attributes. It also include public methods for setting and getting all these attributes.

In the case of sample and synth blocks, they instead count with a **Player Block Attributes** component, which extends the `BlockAttributes` class with the specific functionality needed by these two blocks.

Loop blocks, in addition to the shape and attributes components, include a **Loop Block** component that manages their internal array of blocks.

4.2.2. Communication with Sonic Pi

As mentioned in previous sections of the work, communication between the main application and Sonic Pi is possible thanks to the use of several scripts from the UnityOSC plugin. This library provides a series of scripts which implement classes that abstract developers from the networking code and allows them to easily implement OSC clients and servers to send and receive OSC messages and bundles.

In relation to the main application of this work, both an `OSCHandler` and an `OSCReceiver` instance are created.

The **OSC Handler** works as a singleton instance, and it is used to send the messages previously discussed to the Sonic Pi program. For this purpose, an OSC client is created with the "SonicPi" identifier and the localhost IP address as the destination address. Port **4560** is specified for the client's initialization, which corresponds to the port used by Sonic Pi for OSC message reception [32].

The OSC Handler is able to transform lists of C# objects into OSC format messages and send them to an already initialized client.

When the application is shut down, the OSC Handler sends an "stop" message to Sonic Pi. This is possible because `OSCHandler` extends Unity's `Monobehaviour` class to work as a `GameObject`. This allows the instance to receive a call to the `Quit()` function whenever the application is closed.

The communication between the main application and Sonic Pi's program flows in a bidirectional fashion, as messages from Sonic Pi indicating the advance of an specific loop are expected. In order to listen for Sonic Pi messages, a different approach is taken.

UnityOSC includes a *receiver* class that provides a simple implementation of an OSC listener that stores all the OSC packets received in a previously specified port inside a queue that can be safely consulted.

To handle Sonic Pi messages, the Sonic Pi Manager firsts creates an instance of the `OSCReceiver` class. Then, a server is opened in port **5555** to listen for incoming messages. The receiver class provides two public methods that can be used to track the messages received: `hasWaitingMessages()` and `getNextMessage()`.

The first of these methods returns `true` when message queue size is greater than 0. The Sonic Pi Manager uses its `Update()`[2] method to constantly inquire the receiver for new messages. When at least one message is found in the queue, it gets popped out of the queue and handled as an "advance_loop" message. This message contains an integer value that corresponds to the advancing loop. The loop's advance is handled by the Loop Manager. The Sonic Pi Manager continues handling this messages until the queue is empty.

4.2.3. Addition, modification and removal of blocks

The sounds that the Sonic Pi program produces are visually represented in the main application as a chain of interconnected blocks that resemble the pieces of a puzzle. The user of the application can change the sound loops in real time by adding and removing blocks to the loops, or by modifying already existing blocks.

Changes in any block are registered internally by their loop with an array of Boolean values, indicating whether a block contains changes. When the loop is asked to yield a list of block messages, it consults this array to only provide the messages of the blocks that have been changed.

Addition of blocks

In order to add a block, the user has to open the Block Menu Panel, which displays the possible blocks that can be added to a loop. If the user drags one of the blocks inside a loop block, the application is told to create a new block and add it to the loop.

For this purpose, all the blocks (including the loop block), count with a *Block Drop Handler* script that manages whenever a block is dropped on them by checking the block's Game Object tag[3] and adding a block of the type specified by the tag. To do this, the block calls a function in the Loop Manager instance, `AddBlockToLoop`, passing the loop's index and the index and action name of the new block. The Loop Manager, which has access to the loop-controlling component of each loop, tells the specific loop to add the block. Finally, the loop instantiates a prefab[4] of the type of block specified. Then, it sets it sibling order inside the scene's hierarchy. This is done because the loop blocks are all contained inside a **Horizontal Layout Group**, an object with a special interface component that

[2]The `Update()` function is called once per frame for every `Monobehaviour`-enabled Game Object. More details on the Update function [36] and Unity's event functions execution order [37] can be consulted in the official documentation.

[3]Unity allows to tag Game Objects with specific words. These tags can be compared in execution to identify objects of certain types.

[4]In Unity, Game Objects can be saved as prefabs, allowing to instantiate replicas of the object in the future [38].

lays all of the object's children in an horizontal sequence automatically. The sibling index of the children determines their order in the sequence, so blocks inside a loop would be set to have their own block index (the index inside the loop's block list) as their sibling index. However, the loop block is also included in the layout, so the sibling index of a block is set to be its own internal index plus one.

After setting the sibling order, the loop must set itself as the loop of the block, configure the block appearance to become attach to the loop, and also update the other block indexes in their Block Attributes component.

Finally, it sets the added block and all those that were updated as changed blocks.

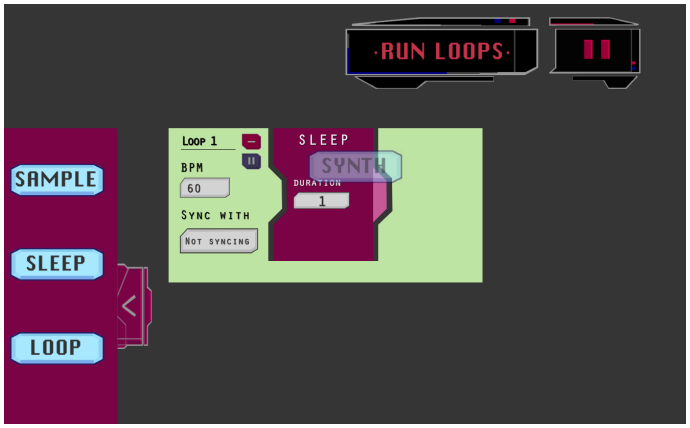

Figure 4.2: An screenshot of the main application where a Synth Block is being added to a loop next to a Sleep Block.

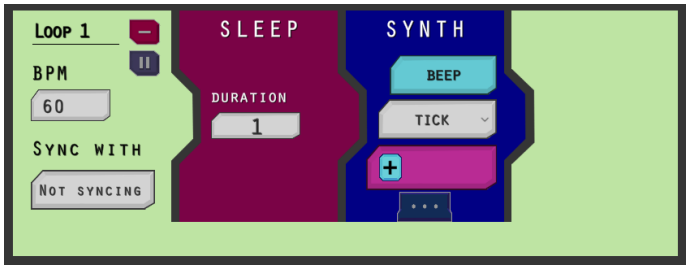

Figure 4.3: An screenshot that shows the effects of the block addition taking place in figure 4.2.

Removing blocks

Blocks can be removed from a loop if they are dragged to a semi-transparent red zone that appears at the bottom of the application when dragging a block.

Removing a block inside a loop indicate that the blocks that were next to it have to be rearranged, both in the application (and visual representation of the loop) and in the Sonic Pi program.

As the way commands are updated in Sonic Pi is by overwriting the command list with the ones that are being modified or added, an issue arise from the removal of a block: the command at the end of the loop would not be overwritten or removed in Sonic Pi, meaning that a duplication of the last command would be played on each iteration.

The approach taken to solve this problem is to add an "empty" command and send it to Sonic Pi. This command, placed at the end of the command list, will be ignored by the processing thread and can be overwritten when the user adds a new block to the loop.

Modifying blocks

The block's command attributes can be changed through input fields in the main application user interface. Sleep blocks contain a "duration" field as their only attribute. The attributes of sample and synth blocks can be edited by opening the corresponding Attributes Panel (by pressing a button in the block's body). These blocks also count with a button displaying the name of the synth or sample being used by the block. When the button is pressed, a panel is opened to display the multiple options for synths and samples that the user can choose. Synth blocks also allow the user to configure the list of notes being played and the playing mode.

When at least one attribute is changed in a block, the loop marks the block as changed, so that its message is send to Sonic Pi to overwrite its attributes.

Moving blocks

Blocks can also be moved within their loops. Moving a block not only implies changing the index of that block, but also rearranging all the contiguous blocks. To simplify this task, a `BlockComparer` class which extends `IComparer` was implemented. The class redefines the `Compare` method to take two blocks and compare their internal index. An instance of this class is used by each loop to sort its list of blocks when a movement takes place.

4.2.4. Addition, modification and removal of loops

As it happens with blocks inside a loop, the loops themselves can also be added and removed by the user. Some properties of the loops can also be edited.

Addition of loops

The application has a maximum of 10 loops that can be managed. If the number of loops is lower than that, the user is allowed to add a new one by dragging the loop-adding element from the lateral menu into a green square inside the Loops Panel. Doing so asks the Loop Manager to create a new loop. The Loop Manager then instantiates a loop prefab as a child of the Loop Container Game Object. The new loop is initialized with its index inside the list of loops and its name is added to a loop name dictionary, used for the selection of loops inside the Synchronization Menu.

Removing loops

When a loop is being removed, the next loops in the list have to be rearranged. This also implies that the internal loop list of the application and the one being played in the Sonic Pi program do not match, what can lead to the duplication of the last loop. This problem is similar to the one that takes place when deleting blocks, and the solution is also analogous: overwrite the last of the loops, which is duplicated, with "empty" commands, turning it into an empty loop.

As some of the loops are overwriting the command list from previous loops in Sonic Pi when rearranging them, it is possible that the overwritten loop contained a larger list of commands than the one taking its place. These extra commands have to also become "empty" commands to avoid inconsistency issues between Sonic Pi execution and its representation in the application.

Another task to perform when deleting a loop is resetting the synchronization of all loops that were synchronized to the one being removed, and finally deleting its name from the name dictionary.

Modifying loops

There are two attributes of loops that can be configured: the BPM (beats per minute) of the loop and the loop to which the loop is synchronizing. The BPM can be introduced in its corresponding input field. For synchronizing one loop to another, there is a drop-down menu that contains the names of all other loops. The user can also change the name of the loops to identify them better, but this will not affect the execution in Sonic Pi.

When the loop attributes are edited, a Boolean variable is set to true. This indicates that when the loop is asked to return its blocks messages, it also has to include an `EditLoopMessage` to overwrite the attributes of the loop being played in Sonic Pi.

4.2.5. Design and implementation of the graphical user interface

The design of the graphical user interface is heavily influenced by Scratch's puzzle-like design. The user adds commands to a sequence by dragging and dropping blocks inside an execution loop. The blocks become visually attach to one another thanks to the edge they have on the right side of their body and the shape-matching notch on the left. All the blocks can be seen as "contained" inside a loop thanks to a bar at the bottom of the block with the color of the loop block and a closing-shape block at the end of the loop.

The art used for buttons and panels was made in collaboration with artist Laura González Pascual. The visual appearance of the interface elements was designed with the intention of giving the application a futuristic aesthetic, with colours inspired by the ones that can be seen in the Sonic Pi editor.

Apart from some "invisible" management objects, the program only consists of interface elements, what in Unity translates to only having visible objects inside of the canvas.

It is important to configure the Canvas Scaler component of the canvas object for the interface objects to resize appropriately depending on the resolution. For this application, the canvas was configured to **scale with screen size**, with a reference resolution of 1920×1080.

Configuration of the blocks shape

In order to implement the puzzle piece appearance of the blocks, they were divided into multiple parts (as seen in figure 4.5).

The main part of every block is the body, which either consists of a rectangular shape

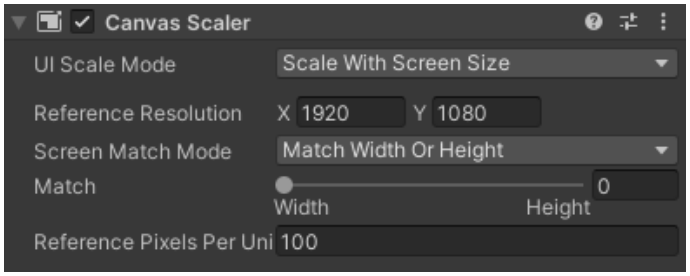

Figure 4.4: A screenshot of the Unity inspector window showing the configuration of the Canvas Scaler for the application.

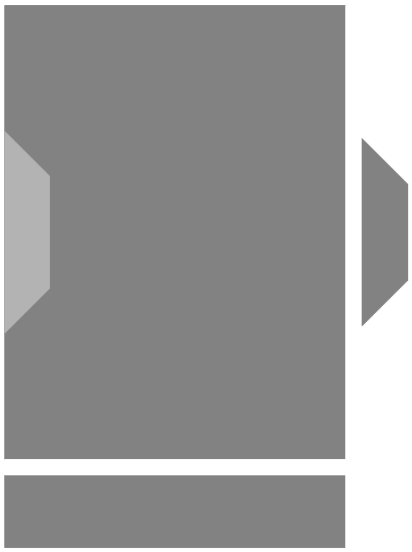

Figure 4.5: A visual representation of the parts that make up a block in the application.

or a rectangle with a notch on the left side. To make the notch appear, the sprite of the body is changed. Inside of the body is where all information and interactive parts of the block will be placed, as well as the indicator that shows up when the block is being played by Sonic Pi.

The bar at the bottom of the block is used to represent it as being contained inside another block (a loop block). When a block is added to a loop, it is configured to add the bottom extension with a color that matches the block's loop color. Apart from the end-of-loop blocks, all blocks are configured to also include a trapeze-shaped edge at the right of the body, which fits with the notch on the left side of other blocks.

To create this block structure in Unity, a set of nested horizontal and vertical layout objects was used.

The first Game Object in each block is the *BlockShape* object, which contains the main components of the block, as well as a Horizontal Layout Group component, that makes its children objects be placed together horizontally and separated by a configurable padding.

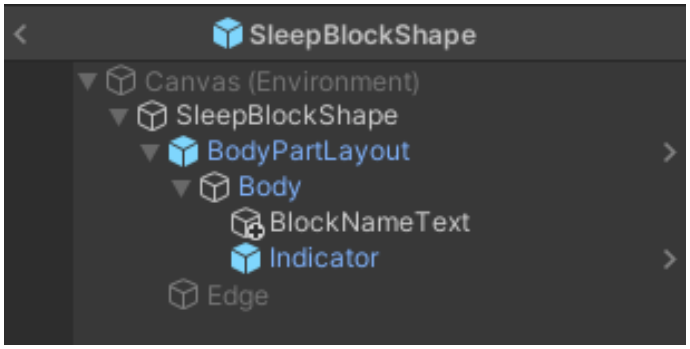

Figure 4.6: A screenshot of the Unity hierarchy window showing the Sleep Block prefab.

The objects contained in this layout are the body (*BodyPartLayout*) and the edge of the block. The *BodyPartLayout* object is a Vertical Layout Group, so it works similarly to the *BlockShape* object but vertically. Initially, it only contains the body of the block, but below the body will be placed the bottom extension that marks the loop that the block is attached to.

In order to allow the configuration of every block's appearance, the `BlockShape.cs` script implements public methods for adding and removing bottom extensions, enabling and disabling the edge and the notch or changing the block's color.

Other elements of the interface

Apart from the loops and blocks, the interface is also composed by other elements that have been previously mentioned. Some of them, like the Run and Pause buttons or the block selection tab are always present, while others appear when the user opens them. This is what happens with the attributes panel and the note, sample or synth selection panels.

In the case of the sample and synth selection panels, as well as the attributes configuration panel, the elements displayed in them are taken from Sonic Pi attributes, synth and samples lists. This makes it possible for the items of the panels to change in future versions of the application, as Sonic Pi may remove or add samples or synths, or start supporting new attributes for samples. To handle this situation, the options displayed on the previously mentioned panels are configured from different JSON files. The content of these files corresponds to the names of attributes, synths and samples present in Sonic Pi. The samples file includes a nested list which divides the catalogue of samples into different categories. To read this kind of JSON file, Unity's JSONSerialize module is not enough, as it doesn't allow to parse JSON files into nested lists. For this purpose, Newtonsoft's Json.NET package [25] was used.

To use external files in a Unity-developed application, it is important to bear in mind that all files not used by the Editor are not included in the final build[5]. This means that a file can not be accessed by code by specifying a file path relative to the Assets folder (the Assets folder does not exist in the build). The easiest way to overcome this issue is to declare a `TextAsset` variable for each file as a `SerializedField`, so the corresponding

[5]Unity does not include the files that are not used by the Editor in the final build in order to reduce the size of the application.

file can be dragged to this variable in the Unity Editor. This avoid the removal of the files in the build, as Unity considers them as Assets.

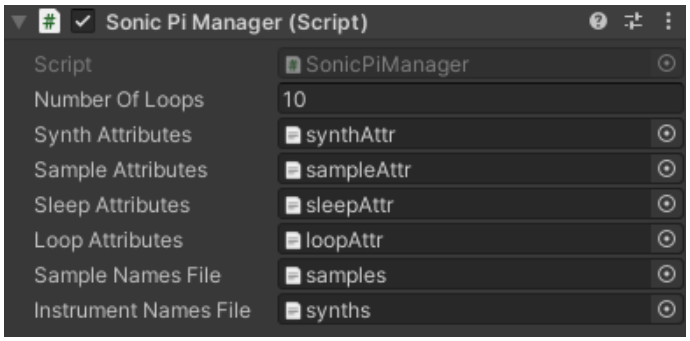

Figure 4.7: A screenshot of the Unity inspector window with the files used in the application attached to the Sonic Pi Manager component.

4.3. Application deployment

As it was previously explained, the application is divided in two programs: the main application and the code that has to be executed with the Sonic Pi. The main application's executable file was generated using Unity's build system. The build is configured to work with Windows, Mac and Linux operating systems[6] as a resizable windowed program with a resolution of 1920×1080.

When the application was first deployed, an error was observed in relation to the configuration of some elements using JSON files. Further investigation of the installation process of the Json.NET package revealed that the application was being built using a .NET version that was incompatible with the package [39]. The issue was solved by changing the build settings in Unity to compile with a more updated version of .NET (see figure 4.8).

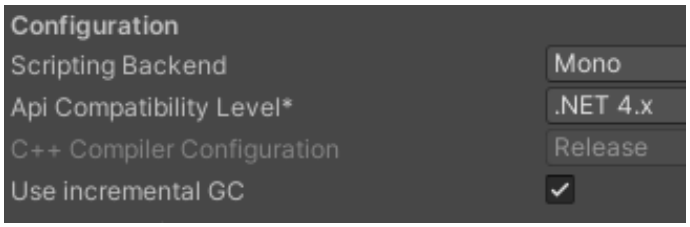

Figure 4.8: A screenshot of the building settings used for the application deployment in Unity.

During the development of the Sonic Pi program, another issue emerged that directly affects the deployment of the application. The issue was noticed when the program stopped working after reaching a certain amount of lines of code. A search in the Sonic Pi official forums determined that a limit with the number of characters a Sonic Pi buffer can send to the Ruby server. The solution suggested by one of the members of the Sonic Pi Core

[6]There might be errors when executing both programs in some operating systems, due to changes in the port used by Sonic Pi for listening to OSC messages. The only operating system in which the application is guaranteed to work is Windows 10.

Team was to execute the command `run_file` next to a string containing the path to the source file that needs to be run.

This solution leaves the application with the need of having two different Ruby source files, one for the source code of the Sonic Pi program and another for the user to load into Sonic Pi and run. Before running the Sonic Pi program, the user must enter the path to the main Sonic Pi program source file. For example:

```
run_file "D:/Applications/SPController/SonicPiController.rb"
```

Another thing to take into consideration is that the user must run the Sonic Pi program before executing the main application in order for the two of them to communicate properly.

Chapter 5

Contributions

This chapter is dedicated to explain the individual contributions of both members of the group to the project.

5.1. Mario Tabasco Vargas

The first stage of the project, Mario was dedicated to the development of the initial **communication system** for the main application to exchange messages with Sonic Pi. This came as a familiar topic to Mario thanks to a project he worked on in the past. For this previous project, he used OSC messages to communicate a game made in Unity with SuperCollider to generate sounds and melodies based on the player's interaction with the game. For implementing this kind of communication, the UnityOSC library was used.

Due to Mario's previous experience with the UnityOSC plugin, in addition to the fact that both projects (the one he worked on and the Sonic Pi Controller application) had a similar approach, it was decided that this would be a good enough way to handle the communication between the main application and Sonic Pi. However, once it was decided the protocol to use, one question remained unresolved: how to turn OSC messages into actual Sonic Pi commands that would generate sound. One decision made in the initial meetings was that we should benefit from as much of Sonic Pi's functionality as possible, specially in regard to the generation of sounds and handling of time and loops. With this approach in mind, Mario started researching the way of turning messages into commands, or even sending actual Sonic Pi's commands for it to execute them. However, he could not find a way to achieve this and instead decided to develop a message structure that would be parsed into Sonic Pi commands.

For the implementation of these **message structures**, Mario considered, among different approaches, to use Ruby classes and inheritance, as different data structures were needed for the attributes of different blocks, but some of them were pretty much the same (specially between synth and sample blocks). However, he stumbled upon an issue in relation to the use of Ruby classes in Sonic Pi (the issue is better described at section 4.1.1). As a consequence of this, and after commenting the issue with Gonzalo and the director,

he decided to use Ruby Structs instead.

After implementing the communication between the two programs and defining the structure of the messages, the next work Mario did on the Sonic Pi program was to implement the **parsing of the OSC messages** and their subsequent **processing** as Sonic Pi commands. For that, he implemented the listening and processing methods that would be executed concurrently inside two different threads. After verifying that both methods were working correctly (although initially not processing all attributes), Mario implemented the **dynamic creation of listening and processing threads**. A pair of these threads is created at the beginning of the application for each loop that the program will handle.

For the main application, Mario started with the development of a simple prototype that sent some command messages for the Sonic Pi program to process, for which he had to implement the classes for the different command structures, as well as a simple drag-and-drop system for adding blocks to a loop. Then, he started working on the interface of the application, starting with the visual appearance of the blocks. For this, the idea was to resemble Scratch's interface, with blocks similar to puzzle pieces that would be attached to a closed piece representing a loop. After Mario implemented all the scripts for the configuration of block shapes, he added the possibility to not only add, but also **move and delete** blocks inside one loop. At that moment, the application allowed to add command blocks to a loop and arrange them as the user likes. However, an issue was found in relation to the sending of multiple messages at once, that resulted in Sonic Pi only treating the first of them. To solve this, Mario made it so all messages from all loops were gathered into a single command structure, which made him modify the Sonic Pi parsing method to adjust to this new message structure.

To add the possibility of modifying any command attributes inside the main application, Mario created **panels for attribute editing and sample/synth type selection**. To configure the options inside these panels, he also added JSON files that are deserialized into dictionaries of attributes and their default values, as well as lists of samples and synth names.

The next big step to take in the development of the application was to implement the **addition and removal of loops**, as up to this moment only one loop was supported. To do this, Mario had to modify both the Sonic Pi program and the `Loop Manager` script. The most complex task when implementing the handling of multiple loops was to properly perform their removal, as this created issues and inconsistencies between the loops in the main application and the ones in Sonic Pi. Mario needed to carry out several tests in order to find and fix all the issues that came up with the handling of multiple loops. He also made it possible for the user to configure and synchronize loops.

Once all the functionality for adding and configuring blocks and loops was working properly, Mario started making **visual improvements** to the main application. He also implemented the **visualization of the block being executed**, by displaying an indicator in the corresponding block and updating the loops by receiving messages from Sonic Pi.

Finally, Mario also worked on the **deployment** of the application and created an installer to simplify the installation process, for which he used the *Inno Setup* tool [40].

For the writing of this paper, Mario was mainly focused on the sections that describe the implementation of the application and the Sonic Pi program, as well as chapter 3, in

which he explained the characteristics of the main technologies that were used during the development process.

5.2. Gonzalo Cidoncha Pérez

When translating a musical Live Coding language into a visual interface, there are two factors to be acknowledged in regards to the user that is going to use it: design and musical coherence. The user might or not have any musical notions or any experience with Live Coding, so the developers have to assure that the design of the application is understandable for all kind of future creators.

Investigating different Live Coding languages is crucial to the development of an application like the one explained in this paper. The developer needs to have some notions about Live Coding and all of its different ramifications, from the ones that are used in live graphics coding, to those more focused in musical performances, giving special attention to this last bunch.

Gonzalo was in charge of exploring multiple Live Coding languages, getting familiar with their syntax and the interface of their environments and studying the features that these languages provided. As a result of this research, Gonzalo was able to identify the main functionalities that a Live Coding application should include. After putting together all these features with Mario, they concluded that the application had to provide users with some specific functionalities that constitute a vital component in the development of any live music coding performance. These group of main features consists on: the potential to add synthesizers that play an arrays of notes, a list of prerecorded samples that the user can play and the opportunity to add intervals of silence and configure the tempo or BPM. Finally, the most important feature that all Live Coding environments include is the incorporation of lists of commands that are executed in an endless loop. It was decided that these loops should become a key component of this work's application, as a loop of commands can be easily represented in a graphical user interface.

After deciding the minimum set features that the application should have, it was time to make a decision on what language to use. There are several musical Live Coding languages that matched the preferences to be had when bringing this idea up. Languages like SuperCollider, FoxDot, Tidal Waves or Sonic Pi are specially attractive, due to their range of possibilities and their first-sight simplicity. SuperCollider was one of the main options that were considered, as it has already serve as a base for a lot of Live Coding software in the past. However, Mario and Gonzalo were not comfortable working with SuperCollider's language *sclang*, so they decided to look for a language that used a more familiar syntax. Sonic Pi, on the other hand, provides a modified version of Ruby (which was more appealing for the authors of this work) and is far more simple to use than SuperCollider, specially for live coding purposes.

With the main language already chosen, how to represent the functioning of said language in a visual interface was the next design process to be discussed. The main design choice was clearly going to be around block-type structures, but how to put them on screen so the user understands how to manage them was really difficult to get to.

When the main application had its communication with the first instances of the Sonic Pi scripts up and running, it was time to start developing the different blocks. The **Synth Block** in particular was the one that had more work behind it, because of the different possibilities it gives to the user. Gonzalo started creating the different elements the Synth Block needed, from the note selection options to the mode the notes were going to be played.

Looking at different music creation applications, there are some different patterns when choosing how to represent the selection of notes. Starting from the notation of said notes, Gonzalo, being mostly familiar with classical music studies, had to choose between the **Alphabetic System** (C to B) and the **Solmization System** (Do to Si). And also, the semitone alteration symbols, the dilemma being if using the *sharp* or the *flat*. Lastly, the choice was the most common of them in other musical software, the Alphabetic System with the usage of sharps when naming altered notes.

The implementation of Synth Blocks consisted of various simple buttons that spawned panel with Vertical Layout Groups for the election of options in the Instrument and the Attributes options. However, when developing the Note Selection list, the functionality of the list was not clear. In a first instance, the idea was to add the note the user wanted straight into the list, but lastly, we both decided too add by default a C4 note each time the user wanted to add a note, in order to let the user choose which note to modify once it is already placed in the list.

With all of the main functions of the application finished, Gonzalo started developing the option of adding a **FX Block**, which gave the sounds that were going to be produced effects like *Reverb* or *Echo*. Sadly, the difficulties given by the implementation of said blocks and the design problems the new feature added made the implementation of this new option very difficult, so we decided to stash them and not release them for the final product.

Finally, Gonzalo was in charge of the **web page** where the application can be downloaded, making it sufficiently coherent with the aesthetic the main application has and caring for it to be understandable with users that wish to download it or contact with us for troubleshoot or any problems we might have not encountered.

Chapter 6

Conclusions and future work

In this chapter, the objectives marked at the beginning of the work will be compared with the actual results of the project. In other words, the chapter analyzes the current state of the application, what it provides to its users and how it adjusts to the initial expectations that were established in chapter 1.

6.1. Conclusiones

As it was explained in chapter 1, the motivation for this project was to introduce less technical users to the concept of live music coding. To achieve this, the idea was to develop an application that would allow users to generate sounds and musical pieces that could be edited in real time, thanks to an intuitive graphical user interface.

Using the Unity video game engine, an application was developed with an eye-catching graphical interface that allows the user to add, delete and modify sequences (or loops) of commands, as well as the commands themselves. This command loops are represented as pieces of a puzzle, that can be attached and rearranged in order to modify the sequence of sounds that the program generates.

The generation of sounds lies with the live coding environment Sonic Pi, which runs the other program developed for the purpose of this work. The application translated the loops that the user creates into OSC messages, which are sent to the other program for it to turn the messages into commands that can be processed by Sonic Pi.

In the following paragraphs, the obtained results will be compared with the objectives that were established at the beginning of the work.

The first of the objectives was to "**provide users with every basic functionality a live coding environment should offer**". Although the complete set of possible functionalities contained in a live coding environment may be unmanageable, the initial intention was to mimic at least the main functionalities found in the Sonic Pi language.

The current version of the application allows the user to: create several loops (up to

ten), configure their BPM, change their names and synchronize them between each other, add sound commands (to play notes lists or samples) using the sounds and synthesizers from Sonic Pi, configure almost all their attributes and add sleep commands.

The functionality currently included in the application is enough to experiment with live music coding and the sounds and synthesizers that are included in Sonic Pi. However, the program lacks some functionalities that are present in Sonic Pi and other live coding environments.

The second of the objectives is related to the **implementation of a graphical user interface to represent, in a clear and visual way, the elements necessary for performing a live coding session**. The application results user-friendly enough for less technical performers, who are capable of creating musical pieces just by dragging and pressing interface elements. In this sense, the use of a block-based graphic interface has proven to be a success when representing sequences of commands. Furthermore, the application allows to visualize the commands of each block that are being executed at any time, which adds a visual characteristic to the application that most live coding environments do not have by default.

Finally, the third objective consisted on "**developing an application that allows users to create artworks that feel different and unique in every session**". Even though this is difficult to evaluate objectively, it may be said that the possibilities that the application offers when creating musical pieces and live coding sessions are enough to allow for the creation of unique works in which users can express their creativity.

In conclusion, we are satisfied with the possibilities that the application offers as a music creation tool for live performances. In our opinion, the usability of the application is good enough to allow artists with no programming skills or even kids to experiment with the tool and enjoy the creation of musical pieces with it. While the application could be use just to experiment with live music creation, we are sure it could be used to introduce users into Sonic Pi and other live coding environments. The application can also prove to be useful for music teachers to show kids a different way of approaching music and introducing them to some basic musical concepts.

6.2. Future work

The application developed for this work can serve as a way to begin in the world of live coding and to familiarize oneself with some of its concepts and how languages like Sonic Pi work. However, it could be said that the idea behind the work has the potential to become a tool that artists and live coders can use in their shows to create music in a more intuitive and visual way (which may result even more attractive to some viewers).

In future versions of the application, the first steps to take would be to implement the functionality not included in the current state of the work that other live coding environments have:

- The possibility to add sound effects (such as echo or distortion) to sequences of commands.

- The possibility to repeat sequences of commands inside a loop.
- The inclusion of special tools that would help to generate rhythmic patterns (for example, Sonic Pi's `spread` command).
- The functionality needed to save the state of one session (with the loop and commands configuration) and load it in future sessions.

The three first tasks correspond to functionalities already included in Sonic Pi, so the way to implement them would be to add new blocks and process them correctly in the Sonic Pi program so that they execute the corresponding command. In the case of the functionality to save and load a session state, a system for saving and loading configurations for loops and commands from a file would be needed.

Another thing to improve in the application would be the installation process, that should be optimized in order to not make the user open both the main application and the Sonic Pi program, as well as preventing him from having to write the path to the program's source file in Sonic Pi.

Regarding the implementation of the application, and more specifically the communication between both programs, it would be a good idea to simplify the message structure in order to also simplify the implementation of new types of blocks.

Chapter 7

Conclusiones y trabajo futuro

En este capítulo se compararán los resultados obtenidos con el trabajo con los objetivos que se establecieron al inicio del mismo. Se analizará el estado final de la aplicación, qué puede aportar a sus usuarios y cuánto se ajusta a las expectativas que se marcaron en el capítulo 1. Finalmente, se expondrán las características que se han quedado sin implementar y que podrían añadirse en futuras versiones de la aplicación.

7.1. Conclusiones

En el primer capítulo de este trabajo se planteaba que la motivación del mismo era acercar a los usuarios sin conocimientos avanzados de programación al mundo de la programación de música en vivo. Para ello, se pretendía desarrollar una aplicación que, mediante una interfaz gráfica de usuario similar a la de otros lenguajes de programación visual, permitiera generar sonidos y piezas musicales que pudieran ser editadas en tiempo real, de manera visual, cómoda e intuitiva.

Utilizando el motor de videojuegos Unity, se desarrolló una aplicación con un aspecto visual llamativo y que permite al usuario añadir, eliminar y modificar secuencias (o bucles) de comandos, así como los propios comandos. En la aplicación, estos bucles de comandos están representados como piezas de un puzle, que se pueden encajar y recolocar para modificar la secuencia de sonidos a producir.

La generación de los sonidos recae en el entorno de live coding Sonic Pi, el cual ejecuta el otro programa desarrollado para este trabajo. La aplicación transforma los bucles creados por el usuario en mensajes OSC, los cuales son enviados al otro programa para que los convierta en comandos que puedan ser procesados por Sonic Pi.

A continuación, se examinará de qué manera se ajustan los objetivos marcados al inicio del trabajo a los resultados obtenidos.

El primero de los objetivos era el de "**proporcionar al usuario la funcionalidad básica que un entorno de live coding debe ofrecer**". Aunque el conjunto de fun-

cionalidades contenidas en un entorno de live coding puede ser inabarcable, al inicio del desarrollo se planteó replicar al menos las principales funcionalidades del lenguaje de Sonic Pi.

La versión actual de la aplicación permite a un usuario: crear distintos bucles (hasta un máximo de diez), configurar sus valores de pulso (o *BPM*), cambiar sus nombres y sincronizarlos entre ellos, añadir comandos de sonido (para reproducir tanto listas de notas como sonidos pregrabados) utilizando los sonidos y sintetizadores de Sonic Pi, configurar casi todos los atributos de estos comandos, además de añadir comandos de silencio (o *sleep*).

La funcionalidad con la que cuenta ahora mismo la aplicación es suficiente para experimentar con la programación de música en vivo y los sonidos y sintetizadores que ofrece Sonic Pi. Sin embargo, se echan en falta algunas de las funcionalidades que sí podemos encontrar en Sonic Pi y otros entornos de live coding.

El segundo de los objetivos estaba relacionado con **implementar una interfaz gráfica que representase de forma clara y visual los elementos necesarios para una sesión de live coding**. La aplicación resulta accesible a usuarios de cualquier perfil, que son capaces de crear piezas musicales solamente arrastrando y pulsando elementos de la interfaz. En este sentido, emplear una interfaz basada en bloques ha demostrado ser un acierto a la hora de representar las secuencias de comandos. Además, la aplicación permite visualizar los comandos de cada bucle que están siendo ejecutados en cada momento, lo que le añade una característica visual con la que no cuentan la mayoría de entornos de live coding de manera nativa.

Por último, el tercer objetivo consistía en "**desarrollar una aplicación que permita a los usuarios crear obras que se sientan diferentes y únicas en cada sesión**". Aunque este punto es difícil de evaluar de manera objetiva, se puede decir que las posibilidades que ofrece la aplicación a la hora de crear piezas musicales y sesiones en directo son suficientes como para crear elaboraciones únicas en las que el usuario pueda expresar su creatividad.

Para concluir, estamos satisfechos con las posibilidades que ofrece la aplicación como herramienta de creación musical para actuaciones en directo. En nuestra opinión, la usabilidad de la aplicación es lo suficientemente buena como para permitir a artistas sin conocimientos de programación, o incluso niños, experimentar con la herramienta y disfrutar creando música con ella. Aunque la aplicación podría ser usada simplemente para experimentar con la creación de música en directo, estamos convencidos de que también podría ser utilizada para introducir a los usuarios a Sonic Pi y al concepto de live coding en general. Además, la aplicación podría resultar útil para profesores de música que quieran enseñar a sus alumnos una manera distinta de acercarse a la música e introducir algunos conceptos musicales básicos.

7.2. Trabajo futuro

La aplicación desarrollada para este trabajo puede servir para iniciarse en el mundo de la programación en vivo y familiarizarse con algunos conceptos del live coding y de

Sonic Pi. Sin embargo, se podría decir que la idea de base tiene suficiente potencial para convertirse en una herramienta que artistas y *live coders* puedan utilizar en sus espectáculos para crear música de una forma más intuitiva y visual (lo que puede resultar más atractivo para algunos espectadores).

En futuras versiones de la aplicación, se debería empezar por implementar la funcionalidad no incluida en la versión actual con la que sí cuentan otros entornos de live coding:

- La posibilidad de añadir efectos de sonido (como eco o distorsión) a secuencias de comandos.
- La posibilidad de repetir secuencias de comandos dentro de un bucle.
- Herramientas especiales que ayuden a generar patrones rítmicos (por ejemplo, el comando `spread` en Sonic Pi, que facilita la creación de ritmo euclídeos).
- La funcionalidad para guardar el estado de una sesión (con la configuración de bucles y comandos) y poder cargarla en el futuro.

Los tres primeros puntos se corresponden con funcionalidades que ya incluye Sonic Pi, por lo que la manera de implementarlos en la aplicación consistiría en añadir nuevos bloques y procesarlos correctamente en el programa de Sonic Pi para que ejecuten el comando correspondiente. En el caso de guardar el estado de la sesión, sería necesario implementar un sistema para guardar y cargar de archivo configuraciones de bucles y comandos, incluyendo los valores de sus distintos atributos.

Otro de los puntos a mejorar de la aplicación sería el proceso de instalación, que convendría automatizar para que el usuario no se vea obligado a abrir tanto la aplicación como Sonic Pi y a introducir en Sonic Pi la ruta del código fuente del programa.

Con respecto a la implementación de la aplicación, y más concretamente a la comunicación entre los dos programas, convendría simplificar la estructura de los mensajes y para facilitar la implementación de nuevos tipos de bloques.

Bibliography

[1] Unity User Manual 2020.3 (LTS), *Unity's interface*. https://docs.unity3d.com/Manual/UsingTheEditor.html, [Last accessed on 2021].

[2] Unity UI 1.0.0 official documentation, *Basic Layout*. https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/UIBasicLayout.html, [Last accessed on 2021].

[3] G. Solis and B. Nettl, *Musical Improvisation: Art, Education and Society*. University of Illinois, 2009.

[4] J. McCartney, "Supercollider: a new real time synthesis language," 2017. http://www.audiosynth.com/icmc96paper.html.

[5] Tidal Cycles main website. https://tidalcycles.org/Welcome, [Last accessed on 2021].

[6] FoxDot main website. https://foxdot.org/, [Last accessed on 2021].

[7] Sonic Pi official website. https://sonic-pi.net/, [Last accessed on 2021].

[8] Pure Data Community website. https://puredata.info/, [Last accessed on 2021].

[9] Unreal Engine 4.26 Documentation, *Blueprint Visual Scripting*. https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/Blueprints/, [Last accessed on 2021].

[10] Blender 2.79 Manual, *Introduction to Nodes*. https://docs.blender.org/manual/en/2.79/render/blender_render/materials/nodes/introduction.html, [Last accessed on 2021].

[11] Scratch main website. https://scratch.mit.edu/, [Last accessed on 2021].

[12] MIT App Inventor main website. https://appinventor.mit.edu/, [Last accessed on 2021].

[13] A. Blackwell, A. McLean, J. Noble, and J. Rohrhuber, *Collaboration and learning through live coding*. University of Cambridge, University of Leeds, Victoria University of Wellington and Robert Schumann Hochschule für Musik, 2013.

[14] M. Wright, A. Freed, and A. Momeni, *2003: OpenSound Control: State of the Art 2003*. Springer, Cham, 2003.

[15] J. K. Haas, *A History of the Unity Game Engine*. Worcester Polytechnic Institute, 2014.

[16] L. Meijer, in the Unity Answers forum. `https://web.archive.org/web/20110414093319/http://answers.unity3d.com/questions/2187/is-unity-engine-written-in-mono-c-or-c`, [Last accessed on 2021].

[17] Unity documentation from the Unify Community Wiki, *UnityScript versus JavaScript*. `https://wiki.unity3d.com/index.php/UnityScript_versus_JavaScript#Overview`, [Last accessed on 2021].

[18] Unity UI 1.0.0 official documentation, *Unity UI: Unity User Interface*. `https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/index.html`, [Last accessed on 2021].

[19] Unity User Manual 2020.3 (LTS), *TextMeshPro*. `https://docs.unity3d.com/Manual/com.unity.textmeshpro.html`, [Last accessed on 2021].

[20] D. Campeanu, *Building UI for games with the new UI Builder - Unite Copenhagen*. `https://www.youtube.com/watch?v=t4tfgI1XvGs`, [Last accessed on 2021].

[21] Unity User Manual 2020.3 (LTS), *The UXML format*. `https://docs.unity3d.com/Manual/UIE-UXML.html`, [Last accessed on 2021].

[22] Unity User Manual 2020.3 (LTS), *Styles and Unity style sheets*. `https://docs.unity.cn/Manual/UIE-USS.html`, [Last accessed on 2021].

[23] Jorge Garcia Martin, in the Open Sound Control (OSC) GitHub repository. `https://github.com/jorgegarcia/UnityOSC`, [Last accessed on 2021].

[24] Unity documentation for version 2020.3, *JsonUtility*. `https://docs.unity3d.com/ScriptReference/JsonUtility.html`, [Last accessed on 2021].

[25] Official Json.NET website by Newtonsoft. `https://www.newtonsoft.com/json`, [Last accessed on 2021].

[26] S. Aaron, A. F. Blackwell, and P. Burnard, *The development of Sonic Pi and its use in educational partnerships: cocreating pedagogies for learning computer programming*. University of Cambridge Computer Laboratory and Faculty of Education, 2017.

[27] S. Aaron interview at CAS TV, *Dr Sam Aaron - Sonic Pi*. `https://www.youtube.com/watch?v=7sEMKXrRaAs`, [Last accessed on 2021].

[28] Ruby language website. `https://www.ruby-lang.org/en/`, [Last accessed on 2021].

[29] S. Aaron, D. Orchard, and A. F. Blackwell, *Temporal Semantics for a Live Coding Language*. University of Cambridge, 2017.

[30] S. Aaron in his Strange Loop Conference talk, *Beating Threads - live coding with real time*. `https://www.youtube.com/watch?v=YlRTTzlhquo`, [Last accessed on 2021].

[31] Sonic Pi official beginner manual. `https://sonic-pi.net/tutorial.html`, [Last accessed on 2021].

[32] Sonic Pi tutorial on OSC handling. https://sonic-pi.net/tutorial.html#section-12-1, [Last accessed on 2021].

[33] S. Aaron in the official Sonic Pi forum, *Is it possible to make custom classes?* https://www.youtube.com/watch?v=TK1mBqKvIyU, [Last accessed on 2021].

[34] Ruby documentation, *Struct*. https://ruby-doc.org/core-2.1.2/Struct.html, [Last accessed on 2021].

[35] Unity documentation from the Unify Community Wiki, *Singleton*. https://wiki.unity3d.com/index.php/Singleton, [Last accessed on 2021].

[36] Unity documentation for version 2020.3, *Monobehaviour.Update()*. https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html, [Last accessed on 2021].

[37] Unity documentation for version 2020.3, *Order of execution for event functions*. https://docs.unity3d.com/Manual/ExecutionOrder.html#UpdateOrder, [Last accessed on 2021].

[38] Unity documentation for version 2020.3, *Prefabs*. https://docs.unity3d.com/Manual/Prefabs.html, [Last accessed on 2021].

[39] A. Davis, *Unity and NuGet + JSON.net*. http://www.what-could-possibly-go-wrong.com/unity-and-nuget/, [Last accessed on 2021].

[40] Inno Setup main website. https://jrsoftware.org/isinfo.php, [Last accessed on 2021].